TNM061 3D Computer Graphics
OpenGL hierarchical transformations
Lab instructions 2016

Stefan Gustavson

February 15, 2019

1 Introduction

This lab exercise is a more or less direct continuation of what you did in last year's introductory course on computer graphics, TNM046. You will be using OpenGL, and you will be taking the same approach to OpenGL programming as last year: learn how to do things from the ground up rather than depend on some higher level API.

Some of the code you will be using has been written for you to make your task possible to finish in just a few hours, but all the code is available for inspection, and we recommend that you take a good look at it. You are not expected to understand every little detail, but you are expected to understand what is done by every function, and how. It's not magic, just code, and it's fairly straightforward code without much complexity.

The code is in C++, but for clarity it is a kind of C++ that does not involve any of the more esoteric features of the language, features we don't really need for the task we have at hand.

The library code is not identical to what you had last year, because it is designed in a proper object oriented manner as classes with methods rather than as a more loosely associated set of functions, but it is very similar. The code from TNM046 did not quite make proper use of classes and methods, but it was designed in an object oriented manner, so the code you see in this exercise should be reasonably familiar to you.

In general, you should not have to worry much about the details. The GLFW dependent code has been written for you, and your task is to set up matrices, shaders and textures and do the OpenGL calls to activate shaders and set shader variables, load and activate textures and draw geometry. This works just like you learned last year, but you will now have some help from having C++ classes. The object orientation makes the syntax easier both to read and to write. Hopefully you will also be a more experienced programmer by now, and be able to focus more on the overview than the nitty-gritty details.

1.1 OpenGL scene management

From the introductory course TNM046, you should know how to do define geometry and draw it, how to write, compile and activate a shader in GLSL, and how to specify and use 4×4 matrices for transformation and projection. However, last year you rendered very simple scenes with one or possibly two objects, and you created and managed the transformation matrix for each object separately. This is inconvenient when the number of objects increases, and an additional level of code is required to make OpenGL more manageable for large and complicated scenes. There are a lot of code libraries to put on top of OpenGL for this purpose, ranging from lightweight helper libraries to complex APIs that sometimes even hide OpenGL altogether from from the programmer's view.

Your experiments here will be limited to three objects, three textures and at most three simple shaders (although the same shader may be used for all three objects), so we will not bother with a more formal management of those assets. For more complex uses of OpenGL, management of scene assets is a key issue that requires proper attention, but for this exercise, we will direct our attention to the management of transformation matrices.

1.2 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.

1.3 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 1.

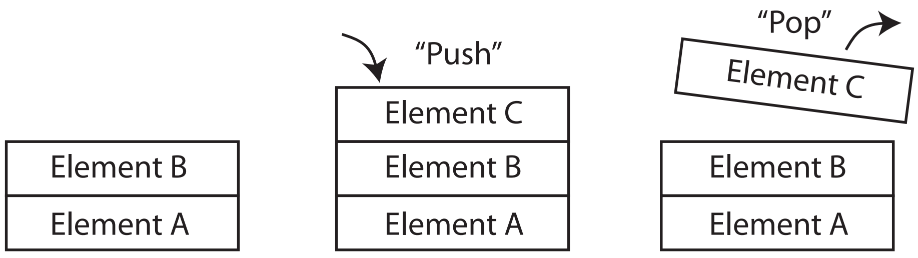

Figure 1: Pushing and popping items on a stack.

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 2, 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”.

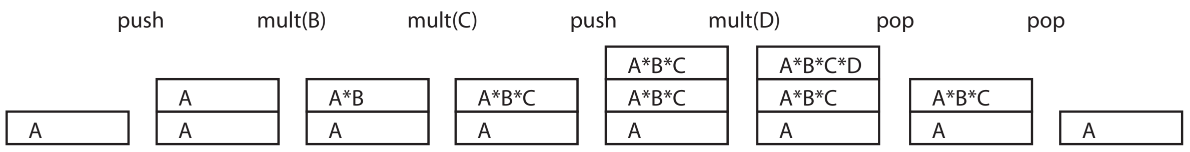

Figure 2: Using a matrix stack.

A class that implements a matrix stack according to the description above is in the C++ file `MatrixStack.cpp` and its corresponding header file `MatrixStack.hpp`. Read the source code for it, 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 on the next page. 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. This is intended for safety checks and debugging purposes. A valid `MatrixStack` object that has a NULL pointer for the field `currentMatrix` has depth 0. A newly created `MatrixStack` has an identity matrix as its only content, and thus has depth 1.

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

1.4 Other library code

The rest of the code which we have already prepared for your convenience resides in five files: `tnm061.cpp`, `Shader.cpp`, `Texture.cpp`, `TriangleSoup.cpp` and `Rotator.ppc`, plus their corresponding header files.

The code in `tnm061.cpp` and `Shader.cpp` together corresponds to what you saw in `tnm046.c` last year: `tnm061.cpp` handles the loading of the relevant OpenGL extensions by the function `loadExtensions()` and provides a function to report the frame rate for the current OpenGL window. `Shader.cpp` defines a data structure to hold information about a shader program, and provides convenient methods to load and compile shaders from files.

In `Texture.cpp`, the only function you should call is `createTexture()`, which loads an OpenGL texture from a TGA file. Only uncompressed TGA files are supported, and the only pixel formats that are recognized are 24-bit RGB and 32-bit RGBA. The TGA file format is old and outdated, but we use it to keep the code small and understandable. A proper OpenGL application would use some modern file format like JPEG or PNG, and use a suitable external library to decode the files.

The file `TriangleSoup.cpp` should be familiar from last year as well. It is an abstraction layer to hide the ugly details around vertex buffers and vertex array objects. The only functions you need to bother with are `TriangleSoup.createSphere()` to create a sphere, and `TriangleSoup.render()` to draw the sphere.

Last, the file `Rotator.cpp` is also an old acquaintance from TNM046, even though it has changed a little to use object orientation. It contains two classes: `MouseRotator` and `KeyRotator`. In the example code, a `MouseRotator` object is used to manipulate the view with the mouse.

These classes are not documented separately and thoroughly, but the source code is reasonably well commented. Furthermore, the intended use of the helper classes is demonstrated by the example program `GLstack.cpp` in the lab material. If you have any further questions, you are of course welcome to ask the lab assistant for help.

2 Exercise: A planet system

In the lab material, you will find an example program `GLstack.cpp` showing the intended use of the library code, along with a Code::Blocks project file to get you started quickly. The example code is heavily commented, and your coding skills should be more than adequate to read, understand and extend it without any further documentation.

Use the provided matrix stack to render a simple hierarchical animated scene: a solar system with a sun, a planet and a moon. Make all the objects textured, and use the stack in the intended manner to keep track of all transformations. Do not use any other transformation matrices in your vertex shader than the current matrix from the matrix stack and the projection matrix. Use a balanced sequence of `push()` and `pop()` so you don't end up pushing more objects on the stack than you remove, because that would make the stack grow in size with each iteration of the rendering loop, and for each frame the memory usage would increase slightly, eventually exhausting the available memory. Granted, it would take hours or even days to fill up the available memory of a modern computer in small chunks of 16 floats, but it is a fundamental program error to not have the rendering loop clean up after itself, and "memory leaks" are a definite no-no in every kind of programming.

Try to make the scene look more or less like Figure 3. The hierarchy is presented as a *scene graph* in Figure 4. Make sure you understand the mapping from the scene graph to the order of operations in your sequential program, and vice versa.

If you like, you can perform `push()` and `pop()` operations for every single matrix you use, but it could become difficult to read your code if you do so. The only time where you really *need* to perform a `push()` is at the scene root (to be able to reset the matrix at the end of the frame) and whenever there is a fork in the tree, i.e. when a node has more than one child node. At those forks, the current transformation needs to be used for more than one branch, with different additional transformations for each branch. For the scene graph in Figure 4, the absolute minimum number of `push()` and `pop()` operations you need is three.

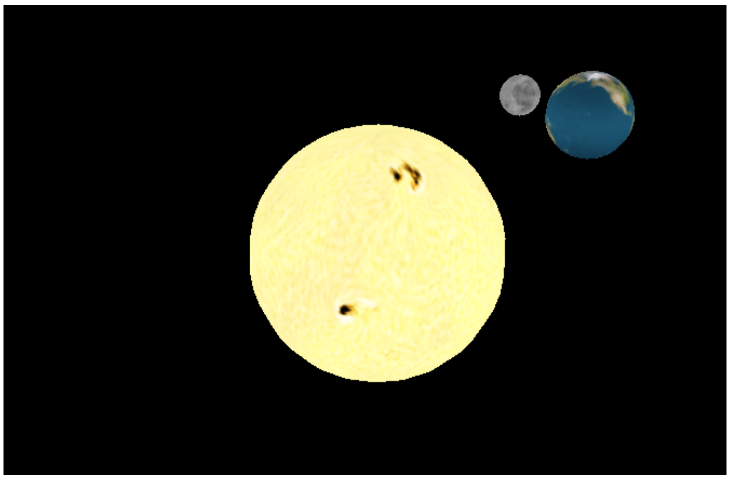
Figure 3: A planetary system with three celestial objects.

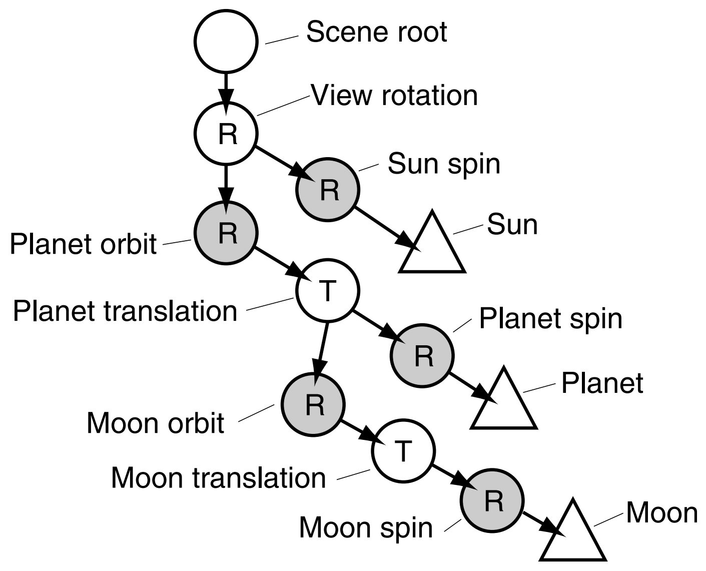

Figure 4: Scene graph for the planetary system. Shaded nodes are animated rotations. White nodes are static transformations.

3 Conclusion

Traversing a scene graph in your own code like this, and handling all user interaction explicitly in the rendering loop, can be a quite reasonable design for smaller programs. For larger and more complex scenes, however, you would want to abstract away many of the rendering details by using a more competent API on top of OpenGL. Such an API could be quite small and simple, and there is nothing to stop you from writing one yourself, but there are also many existing APIs to choose from. At the heart of it, though, there is still OpenGL code, and it is extremely useful to know what is under the hood. Sometimes you really need to bother with the details in order to do what you want.

OpenGL is a useful graphics API. It has kept up with the rapid development in the field since it was first introduced in 1992. The introduction of OpenGL 3.0 in 2008 made some big and fundamental changes, but it was definitely for the better. Contrary to many other graphics APIs, OpenGL has managed to strike a good balance between introducing modern functionality and maintaining reasonable stability between versions, and it will most likely continue to be useful for quite some time yet.