TDT4195: Visual Computing Fundamentals

Computer Graphics - Assignment 2

October 11, 2016

Bart van Blokland
Department of Computer and Information Science
Norwegian University of Science and Technology (NTNU)

- **Delivery deadline: October 28th, 2016 by 22:00.**
- **This assignment counts towards 4% of your final grade.**
- You can work on your own or in groups of two people.
- Deliver your solution on *itslearning* before the deadline.
- Use the Gloom project along with all modifications from Graphics Lab 1 as your starting point.
- Do not include any additional libraries apart from those provided with Gloom.
- Upload your report as a single PDF file.
- Upload your code as a single ZIP file solely containing the *src* and *shaders* directories found in the Gloom project. All modifications must be done to files in these directories.
- Your code must be runnable on the lab computers at IT-S 015.
- All tasks must be completed using C++.
- Use only functions present in OpenGL revision 4.0 Core or higher. If possible, version 4.3 or higher is recommended.
- The delivered code is taken into account with the evaluation. Ensure your code is documented and as readable as possible.

Questions which should be answered in the report have been marked with a [**report**] tag.

Objective: Understand how multiplying a 4x4 matrix with single coordinates allows for a wide variety and combinations of transformations. Get a taste of combining transformations, and what the limits are of their possibilities.

In the previous assignment we've seen how to draw geometric primitives using OpenGL, and had a taste of GLSL Shaders. This assignment focuses on affine transformations, and their effects on a scene.

This assignment starts from the point where assignment 1 ended. As such you should use the code from the previous assignment as your starting point for this one.

Task 1: Repetition [0.5 point]

a) **[0.3 point]** In assignment 1, you created a function in Task 1a which converted an array of vertices and an array of indices into a Vertex Array Object (VAO).

   i) Extend this function to in addition take another float array as a parameter. This additional array should contain colour values - one RGBA colour value per vertex. This additional float array should be put into a Vertex Buffer Object, and attached to the VAO.

   Hint: this should only be a matter of copying and pasting sections of the original function and modifying some parameters.

   Hint: colours in OpenGL usually consist of 4 floats. Each float represents one channel in the order [red, green, blue, alpha]. The value of each channel should be within the range [0, 1], per OpenGL's specification.

   The alpha channel denotes the transparency of the colour. A value of 1 denotes complete saturation, while 0 will generate a colour which is entirely transparent.

   ii) Modify the Vertex ("gloom/shaders/simple.vert") and Fragment ("gloom/shaders/simple.frag") shaders to use colours from the new colour buffer as the vertex colours of triangle(s) in the buffer. If vertices in the same triangle have been assigned different colours, colours should be interpolated between these vertices.

   Hint: both shaders require modification to accomplish this.

b) **[0.2 point]** **[report]** Render a scene containing at least 5 different triangles, where each vertex of each triangle has a different colour. Put a screenshot of the result in your report, along with the definitions of your vertex, colour, and index arrays. All triangles should be visible on the screenshot.

Task 2: The Affine Transformation Matrix [0.8 point]

a) [**0.2 point**] Modify your Vertex Shader so that each vertex is multiplied by a 4x4 matrix. The elements of this matrix should also be defined in the Vertex Shader.

Change the individual elements of the matrix so that it becomes an identity matrix. Render the scene from Task 1b to ensure it has not changed.

Note that matrices in GLSL are column major. As such, individual elements in a matrix can be addressed using:

```
matrixVariable[column][row] = value;
```

It is also possible to assign a vec4 to a column, to set all 4 values at once:

```
matrixVariable[column] = vec4(a, b, c, d);
```

b) [**0.6 point**] [**report**] Individually modify each of the values marked with letters in the matrix in equation 1, one at a time. In each case use the identity matrix as a starting point.

For each of the elements marked in equation 1, describe:

i) Observe the effect of modifying the value on the resulting rendered image.

ii) Deduce which of the four affine transformation types discussed in the lectures and the book modifying the value corresponds to.

iii) Write down the direction (axis) the transformation applies to.

iv) Briefly describe how, when multiplying the matrix with a 4x1 coordinate vector, the rules of matrix multiplication causes the particular affine transformation to happen.

$$\begin{bmatrix} a & b & 0 & c \\ d & e & 0 & f \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

Hint: you can use a uniform variable to pass a floating point value slowly oscillating between -0.5 and 0.5 from the main loop in program.cpp into the Vertex Shader. Use an accumulating variable which is slightly incremented each frame in the main loop, and calculate the sine of it before passing it into the Shader.

It might make the effects of modifying individual values in the transformation matrix much easier to recognise. The guide includes a description on how to work with uniform variables.

Task 3: Combinations of Transformations [2.1 point]

The true power of affine transformations matrices lies in their ability to combine any number of transformations in any order into a single transformation matrix.

This makes applying a large number of complex transformations on a large scene extremely cheap compared to applying each transformation individually. You always need to multiply a single matrix with a given coordinate, as opposed to applying each transformation individually.

This task aims to show both the power and limits of which transformations can be achieved with a single 4x4 transformation matrix.

The first objective of this task is to develop a working, controllable "camera", suitable for viewing scenes.

For the duration of this task, using the functions provided by, or copying code from Gloom's *Camera.hpp* is not allowed.

a) **[0.4 point]** Alter your Vertex Shader so that the transformation matrix by which input coordinates are multiplied is passed in as a uniform variable. The guide contains information on how to accomplish this.

   Alter the main loop to set the value of the created uniform variable each frame, prior to rendering the scene.

   Hint: Shader Programs must be activated (used) before you can change the values of any of their uniform variables.

   Hint: You can use GLM's `glm::value_ptr(glm::mat4x4 matrix)` function to generate a data pointer do the transformation matrix's contents to a value pointer you can pass into the `glUniformMatrix4fv()` function.

   Hint: You can define a new variable and initialise its value to an identity matrix using `glm::mat4x4 someMatrixVariable(1.0);`.

b) **[0.3 point]** Implement key press event handlers which (in a future task) allow the camera to be:

   i) Moved left or right
   ii) Moved up or down
   iii) Moved forward or backward
   iv) Rotated horizontally
   v) Rotated vertically

Store the camera location and orientation in variables. For simplicity, it may be easiest to store these in global variables (although this is usually not considered good practice).

The camera should behave as a camera on a tripod or a human head. That is, it should allow horizontal and vertical rotation of the camera's point of view relative to the scene. In addition, it should be possible to translate the camera along the x-, y-, and z-axis.

You are free to define yourself which keys are bound to which action.

It is sufficient to make the camera move aligned along the major axis. It is not necessary to make all movement relative to the direction the camera is facing, although you are free to do so.

Hint: Gloom's *program.cpp* already has a predefined a keyboard callback you can utilise, which contains a single keystroke handler. Adding additional keystroke handlers can be done by duplicating the existing if statement, and changing the constant testing for which key was pressed. A list of all supported keys can be found here:

`http://www.glfw.org/docs/latest/group__keys.html`

c) [**0.9 point**] Based upon the stored location and orientation of the camera defined during the previous subtask, implement a sequence of transformations which rotates and translates the camera as described in the previous subtask.

The resulting transformation matrix should be passed into the Vertex Shader as the transformation matrix (using the uniform variable you set up previously).

Hint: consider whether there is a difference between rotating a camera inside a stationary world, or rotating a world around a stationary camera.

Hint: GLM has some useful functions for generating various kinds of transformation matrices. Here is a listing of the functions which may be relevant for this task:

- `glm::mat4x4 glm::rotate(float angle, glm::vec3(float x, float y, float z));`

  The x, y, and z parameters should represent a normalised vector. Generates a 4x4 transformation matrix representing a rotation around the defined vector.
- `glm::mat4x4 glm::translate(glm::vec3(float x, float y, float z));`

  Generates a 4x4 transformation matrix representing a translation by $[x, y, z]$ units.
- `glm::mat4x4 glm::scale(glm::vec3(float x, float y, float z));`

  Generates a 4x4 transformation matrix representing a scale transformation by a factor of $[x, y, z]$. Remember that scaling by a factor of 1 leaves the scene intact.

Note that all functions require passing in glm::vec3 structures in some way. These have been added and expanded for clarity on how to use them. All functions return an instance of glm::mat4x4.

Hint: GLM's matrices and vectors have predefined arithmetic operators. As such you can multiply, add, subtract, and divide matrices in the same way you would multiply two numbers together (for instance: `matrix3 = matrix1 * matrix2`).

Hint: the order in which you multiply matrices is key to achieving the desired behaviour of the camera. Try to deduce a rule regarding the order in which matrices are multiplied together and the order in which those transformations appear to be applied to your scene. If you do, start with two-dimensional transformations before moving on to three dimensions, because they are a lot easier to work with.

Hint: the following includes should be most or all of the ones you need to complete this task:

```
#include <glm/mat4x4.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <glm/gtx/transform.hpp>
#include <glm/vec3.hpp>
#include <glm/gtc/matrix_transform.hpp>
```

d) [**0.5 point**] [**report**] The figures in this question show pairs of geometric structures. In each pair, one more more objects are shown, as well as the same object(s) having undergone some transformation.

Coordinates are shown where relevant to indicate scale. Determine in each case whether the input object(s) can be transformed by a single 4x4 transformation matrix to produce the shown transformed object.

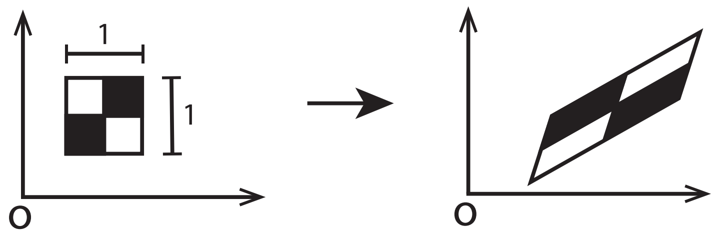
If the transformation is possible, give a sequence of transformations you would apply in order to achieve the desired shown transformation. Specifying exact dimensions, values, angles, and so on is not required. Giving the transformation type (affine or otherwise) and axis in each step is enough.

If you can't express the transformation in terms of basic transformation types, describe change(s) to specific fields in the 4x4 matrix which would make it possible.

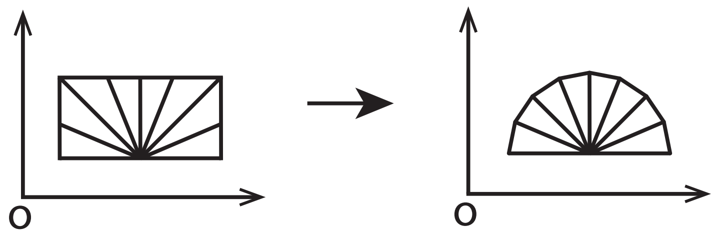
If you think the transformation is impossible, give reasons why.

All coordinates of the shown shapes lie on the xy-plane (all z-coordinates are 0).

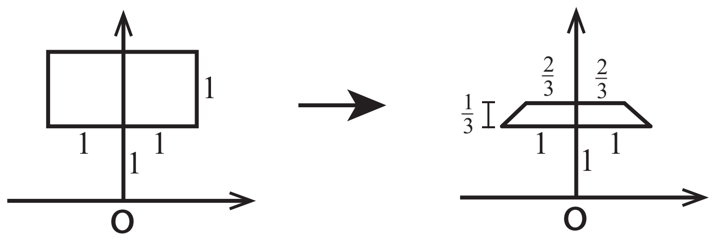
i)



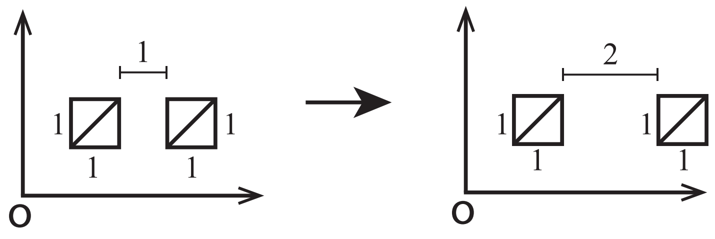
ii)



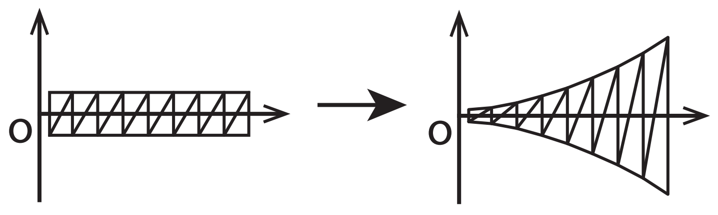
iii)



iv)



v)



Task 4: Projections [0.6 point]

When you look around you, an object which is very close to you appears to be much larger than when the same object would be far away. This is caused by what is referred to a *projection* of the world around you on the retina of your eye.

This effect is caused by the way in which light rays reach your eye. When an object is closer, it will relatively occupy more space in your field of vision, and thus appear larger.

If it is desirable to render a scene in three dimensions convincingly for us (humans), this effect must be simulated. Fortunately, as shown in the lectures this can also be achieved through a single 4x4 transformation matrix similar to the ones we've been working with thus far.

In a sequence of transformations, the projection should be the final one that is applied on the object(s) being rendered.

a) **[0.4 point]** Apply the projection described in the introduction of this question on your scene so that it appears to be in a three dimensional scene which looks convincing to humans. You may need to add a rotation and translation transformation to ensure the effect is visible, and you scene remains within view (translating your scene along the z-axis should work).

   You are not required to define this matrix manually, as GLM has two handy functions to generate matrices for the two most common projection types you can use:

   i)
```
glm::mat4x4 glm::perspective(
        float FOVRadiansY, float aspectRatio,
        float nearPlane, float farPlane);
```

   Generates a transformation matrix representing a perspective projection. The first parameter specifies the Field Of View (FOV) the *camera* is capable of capturing. The second parameter specifies the aspect ratio of the window, defined as the window width divided by the window height.

   Finally, the nearPlane and farPlane parameters specify how far the near and far clipping planes of the frustrum should be located away from the origin. It is recommended you set these to 1.0 and 100.0, respectively.

   ii)
```
glm::mat4x4 glm::ortho(
        float left, float right,
        float bottom, float top,
        float zNear, float zFar);
```

   Generates a transformation matrix representing an orthographic projection. The parameters represent the boundaries of the clipbox. For instance, passing in 0 and 100 into the left and right parameter, respectively causes all x-coordinates between 0 and 100 to be in view of the camera.

b) **[0.2 point]** **[report]** Put a screenshot in your report showing the results of the previous task, clearly showing the effect which the projection has on your scene.