TDT4195: Visual Computing Fundamentals

Computer Graphics - Assignment 2

September 10, 2020

Michael Gimle
Bart van Blokland

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

- **Delivery deadline: September 18th, 2020 by 23:59.**
- **This assignment counts towards 5% of your final grade.**
- You can work on your own or in groups of two people. Please note that we have significantly higher expectations in this case.
- Deliver your solution on *Blackboard* before the deadline.
- Use the Gloom-rs 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-rs.
- Upload your report as a single PDF file.
- Submit your code folder as a .zip, making sure to not include the target directory.
- All tasks must be completed using Rust.
- 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.
- Currently MacOS support for OpenGL is spotty at best. We strongly advise you to use Windows or Linux instead, if at all 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 lab we've seen how to draw geometric primitives using OpenGL, and had a taste of GLSL Shaders. This lab focuses on affine transformations, and their effects on a scene.

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

Task 1: Repetition [0.5 points]

a) **[0.3 points]** In Lab 1, you created a function in Task 1a which converted a float (f32) and integer (u32) vector into a Vertex Array Object (VAO).

   i) Extend this function to in addition take another float vector as a parameter. This additional vector should contain colour values - one RGBA colour value per vertex. This additional float vector 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. Bear in mind that the attribute pointers need to be specified while the buffer they're referring to is bound.

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]. In each channel, a value of 1 denotes the maximum value, while 0 represents the minimum value.

The alpha channel denotes the transparency of the colour. In this case, a value of 1 represents an entirely opaque colour, while a value of 0 means the colour is completely transparent.

In Gloom-rs, transparency has been turned on by default. This is something that has to be explicitly enabled in an OpenGL project, and is done with the following lines::

```
gl::Enable(gl::BLEND);
gl::BlendFunc(gl::SRC_ALPHA, gl::ONE_MINUS_SRC_ALPHA);
```

The `gl::Enable()` function enables a particular OpenGL feature. In this case, the parameter `gl::BLEND` specifies that you'd like to turn on alpha blending. The `gl::BlendFunc()` specifies how a transparent colour should be mixed with colours from objects behind it. The values shown above are usually the ones you'll want to use.

Blending in OpenGL is done by looking at the colour currently present in the frame-buffer, as well as its transparency, and combining it with the new colour and its transparency as defined by the blending function specified in `gl::BlendFunc()`.

In this case, a new colour is calculated like this:

$$NewColour = (SourceAlpha) \cdot SourceColour + (1 - SourceAlpha) \cdot DestinationColour \quad (1)$$

The source colour and alpha are the colour and transparency of the new pixel being drawn, respectively. The destination colour is the colour of the pixel already present at the same location in the framebuffer.

Unless you want to achieve some very specific effects, the `gl::BlendFunc()` parameters shown above are almost always the ones you'll want to use.

ii) Modify the Vertex ("gloom-rs/shaders/simple.vert") and Fragment ("gloom-rs/shaders/simple.frag") shaders to use colours from the new colour buffer as the 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 points]** **[report]** Render a scene containing at least 3 different triangles, where each vertex of each triangle has a different colour. Put a screenshot of the result in your report. All triangles should be visible on the screenshot.

Task 2: Alpha Blending and Depth [0.5 points]

a) **[0.2 points]** **[report]** For this task, we want to show you what happens to a blended colour when multiple triangles overlap from the camera's perspective (where the triangles are positioned at different distances from the camera).

To this end, draw at least 3 triangles which satisfy the following conditions:

- There exists a section on the xy-plane on which all triangles overlap.
- For any single triangle, the three vertices of that triangle have the same z-coordinate.
- No two triangles share the same z-coordinate.
- All triangles have a transparent colour (alpha < 1).
- All triangles have a different colour.
- Each triangle's vertices have the same colour.

I do not recommend drawing the exact same triangle 3 times at different depths; it will make the next question more difficult to solve.

Remember to turn on alpha blending, as described in the previous task.

Put a screenshot in your report.

b) **[0.3 points] [report]** First make sure your triangles are being drawn back to front. That is, the triangle furthest away from the screen is drawn first, the second-furthest one next, and so on. You can change the draw order of triangles by modifying the index buffer. Remember the coordinate space of the Clipbox here.

i) Swap the colours of different triangles by modifying the VBO containing the colour Vertex Attribute. Observe the effect of these changes on the blended colour of the area in which all triangles overlap. What effects on the blended colour did you observe, and how did exchanging triangle colours cause these changes to occur?

ii) Swap the depth (z-coordinate) at which different triangles are drawn by modifying the VBO containing the triangle vertices. Again observe the effect of these changes on the blended colour of the overlapping area. Which changes in the blended colour did you observe, and how did the exchanging of z-coordinates cause these changes to occur? Why was the depth buffer the cause this effect?

Task 3: The Affine Transformation Matrix [0.7 points]

a) **[0.2 points]** 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);
```

Refer to the guide for more information.

b) **[0.4 points] [report]** Individually modify each of the values marked with letters in the matrix in equation 2, one at a time. In each case use the identity matrix as a starting point.

Observe the effect of modifying the value on the resulting rendered image. Deduce which of the four distinct transformation types discussed in the lectures and the book modifying the value corresponds to. Also write down the direction (axis) the transformation applies to.

$$\begin{bmatrix} a & b & 0 & c \\ d & e & 0 & f \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

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 main.rs into the Vertex Shader. Use the elapsed variable in the main loop, and compute the sine of of it (`elapsed.sin()`) before sending it into the shader with a uniform variable. See the guide for how to do this.

It makes 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.

c) **[0.1 points]** **[report]** Why can you be certain that none of the transformations observed were rotations?

Optional Recommended Intermediate: Play around with the i3t-tool [0 points]

The task after this one requires you to implement a camera with multiple axis of freedom. Games usually implement this as a combination of transformations.

You've seen in the previous task how a single matrix can cause a number of deformations to occur to a shape as a direct result of the behaviour of matrix multiplication. But that's not what makes these matrices so amazing.

The main thing that causes affine transformations to be used ubiquitously in games nowadays is because of the following equation:

$$(A \cdot B)v = (A \cdot (Bv)) \tag{3}$$

Here v is a given vertex inside your model, while A and B are transformation matrices. While the difference may appear insignificant, the important thing here is that A and B can be precomputed. A and B can even be a much longer sequence of arbitrary transformations if desirable. It is not necessary to multiply each matrix with each vertex individually; something which would take an unholy amount of time even on good hardware.

But there's also a conceptual side to this. If I have a long sequence of matrices, each representing some sort of transformation:

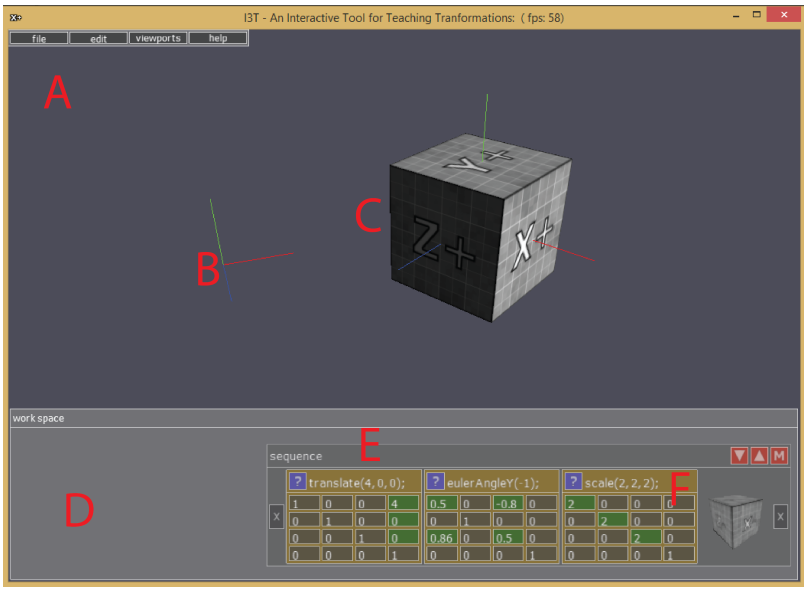


Figure 1: The main window of the i3t tool.

$$v' = (A \cdot B \cdot C \cdot D \cdot E \cdot F \cdot G \cdot H)v \tag{4}$$

The *order* in which they are multiplied *matters*.

This is due to the rules of matrix multiplication not being commutative. In order to gain an understanding of how the order of multiplication affects the resulting transformation, I *highly* recommend checking out a tool called "I3T", which you can download from here:

`http://i3t-tool.org/`

Unfortunately, since this is a Windows application, we can't install them on the lab machines. In Figure 1 you can see a screenshot of what the program looks like when you start it initially.

The area marked with A is where you can view the scene you've constructed. B is the scene's origin (the x, y, and z-axis are marked with a red, green, and blue colour, respectively).

The scene initially contains a single cube object (C). As you can see, it has its own coordinate system, which has undergone some transformations. These transformations are shown as yellowish boxes (F) in the scene editing area (D).

There are three transformation matrices present in the area which defines the "sequence" of transformations applied to the cube present in the scene (E). Note that the transformation type is listed in the header of each matrix.

You can add a new transformation by right clicking somewhere in the vicinity of D, click "transformation", and select the type you'd like to add. Next, you can drag the new matrix that appears into the existing sequence. You can also reorder them this way.

To modify matrices, notice that some fields are highlighted with green. You can drag these to incrementally modify the transformation (such as the angle in case of rotations). For a given sequence of transformations, the matrices applied on the object are multiplied together in the order in which they are shown.

Interlude: Transformation functions in glm

Here are the (simplified signatures of) glm functions you'll want to use to produce the various transformation matrices, and examples of how you'd use them:

- ```
  fn glm::rotation(angle: f32, axis: &glm::Vec3) -> glm::Mat4;
  ```

  ```
  let rotation: glm::Mat4 = glm::rotation(angle, &glm::vec3(1.0, 0.0, 0.0));
  ```

  `axis` should be a normalized vector representing the vector around which to do the rotation. The angle is in radians.

- ```
  fn glm::translation(direction: &glm::Vec3) -> glm::Mat4;
  ```

  ```
  let translation: glm::Mat4 = glm::translation(&glm::vec3(0.0, 1.0, 0.0));
  ```

  Generates a 4x4 transformation matrix representing a translation by $[x, y, z]$ units.

- ```
  fn glm::scaling(scale: &glm::Vec3) -> glm::Mat4;
  ```

  ```
  let scaling: glm::Mat4 = glm::scaling(&glm::vec3(1.0, 1.0, 1.0));
  ```

  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.

- ```
  let identity: glm::Mat4 = glm::identity();
  ```

  Generates a 4x4 identity matrix.

Task 4: Combinations of Transformations [3.3 points]

Please do not be put off by the size of this question. The main reason for there being so much text is to ensure the objective of the question is clear, and to explain some concepts surrounding it. For reference, when I implemented this question, I needed around 30 lines of code. Most of which was a matter of copying and pasting.

We've now seen some basic affine transformations. However, we have not yet looked at what makes them so useful and powerful: their ability to combine. To show this, consider a point p, and two transformations which we wish to apply on this point; A and B. Here p is a 4x1 matrix, and A and B are both 4x4 matrices. We can transform p by A, simply by performing matrix multiplication on them: $A \cdot p$. Since the result of this multiplication is again a 4x1 matrix, we can multiply the result with B to get a transformed point p': $p' = B \cdot (A \cdot p)$.

The key point here is that matrix multiplication is *associative*. That is, we don't necessarily need to multiply matrices in the order shown above. Instead, we can multiply A and B first, which yields another 4x4 matrix, and multiply this result with point p. This 4x4 matrix represents the combined result of the transformations A and B. Multiplying this matrix with p will yield the same point as before p'.

We can thus multiply any number of transformation matrices together and still always be left with a single 4x4 matrix which represents the entire sequence of transformations.

Why is this such an important property? It means that we can apply any number of affine transformations on a point by multiplying it with a single 4x4 matrix, rather than having to compute the result of each individual transformation separately for every single vertex. This saves a massive amount of computation in the vertex shader each frame.

We'll now implement the next major component of our animated scene: a controllable camera. We'll use combined affine transformations to accomplish this.

a) **[0.4 points]** 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. An important thing to note is that when passing a matrix from glm to the OpenGL uniform functions, you want to use `.as_ptr()` on the matrix.

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

   For the remainder of this task, do all creation and multiplication of matrices on the CPU (not in the vertex shader), and pass on the resulting combined 4x4 matrix through the uniform variable you created.

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

Note: If your OpenGL version is below 4.3, you will *not* be able to use the layout qualifier to explicitly set the index of your uniform variables, and thus you will have to query the shader for the positions of uniforms. See the newest version of gloom-rs to find a helper method for getting this location.

b) **[0.4 points]** The first transformation we'll apply is a projection. Specifically, we'll apply a perspective projection so that our scene looks more comparable to a "real world" scene. Fortunately, the GLM library which has been included with Gloom-rs contains a helpful function for generating such a transformation matrix:

Any time you use any of the functions from the glm library, you should be specifying the type for the resulting variable as `glm::Mat4`. This is a 4x4 matrix consisting of f32, which will help you avoid any nasty surprises down the line if the type system concludes you want a double (f64) but you tell OpenGL you're using floats. This is necessary because the glm matrix functions are implemented as generic over matrix dimensions and datatypes, and OpenGL doesn't help Rust do its type inference.

```
let variable_name: glm::Mat4 = glm::perspective(
    aspect: f32, fovy: f32,
    near: f32, far: f32 );
```

The first parameter of this function specifies the aspect ratio of the window, defined as the window height divided by the window width. The second parameter specifies the vertical Field Of View (FOV) the *camera* is capable of capturing. That is, the angle between the top and the bottom plane of the view frustum.

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

If you do, any triangles whose coordinates are near the $x = 0$ and $y = 0$ mark (which your triangles from task 1 already are) will be visible between z-coordinates ranging between -1 and -100. Why, you might ask? You see, the projection matrix return by `glm::perspective()` *flips the z-axis.*

Therefore, the coordinate system of the Clipbox, which is a left-handed one, is converted into a right-handed one. There's not much logic to this other than that it's an OpenGL convention. This means that the order of which triangles are closest to the screen will get reversed.

Therefore, in order to both draw your triangles with a perspective projection, and ensuring they don't go out of view, do the following:

- Apply a translation transformation on the matrix you send into the vertex shader. Translate the triangles into the negative z direction. Note that the

visible coordinate range of triangles lies between -1 and -100, assuming you went with the near and far plane distances we suggested. At this point, your triangles should become invisible.

- Now apply the perspective projection on the transformation matrix sent into the vertex shader. Make sure it's the last transformation applied on the triangles. You can now compile and run your program, and your triangles should be visible again.

Creating the Camera

Before we continue, let us first consider some important aspects to understand for creating a camera. The most important of which is that a projection effectively only changes the shape and dimensions of the Clipbox. In a way, the projection causes the contents of an arbitrary frustum volume to be morphed into the shape of the Clipbox.

As such any geometry located inside of the view frustum defined by the projection is inserted into the rendering pipeline and processed. Anything outside of it is clipped away.

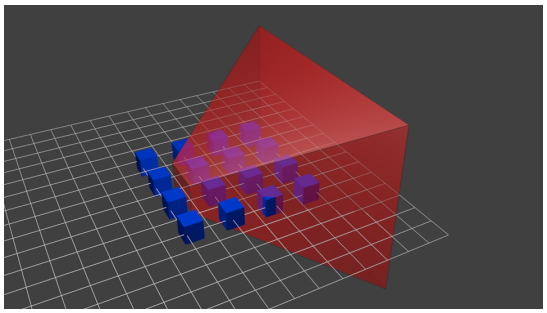
For instance, an Orthographic projection will linearly scale coordinates along each axis to fall into the range [-1, 1]. The perspective projection is a bit more complicated, and is shown in Figure 3.

There are two important observations to make here. First, projections define their own origin, view frustum, and scale. Knowing where your origin is located is important because affine transformations are applied relative to this origin. For instance, a rotation transformation causes an object to be rotated around the origin in a particular direction.

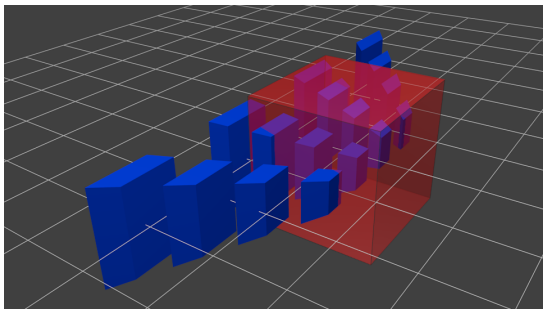
Scale is also relevant, because it determines the size the objects you are rendering need to be. For instance, using an Orthographic projection to create a frustum with a height of 100 units will cause a model with a height of 1000 units to almost be clipped away in its entirety, because it does not fit in the view frustum.

The second and more relevant observation revolves around the notion that the view frustums defined by projections can not moved around. For instance, the perspective projection is dependent on being "located" at the origin due to its mathematical definition.

Since the purpose of a camera is to be able to move through a scene and view it from different angles and positions, but the view frustum of the camera cannot be moved around, we instead have to *move the entire world around the camera*. In other words, our "camera matrix" needs to translate the world in such a way that the camera's position becomes the origin, and rotate it such that the world's x, y and z axes becomes aligned with the camera's orientation. The end result is that anything that is directly in front of the camera will end up laying along the z-axis. This has been illustrated in Figure 2.

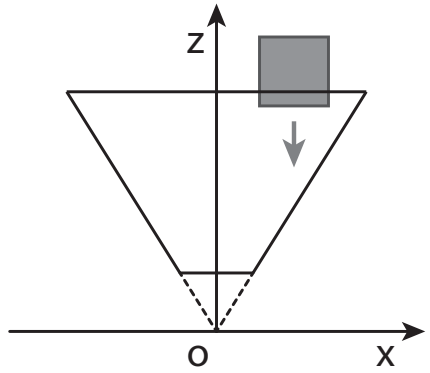


(a) The perspective projection view frustum containing a scene of cubes.

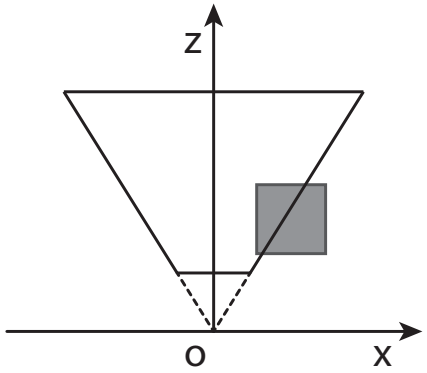


(b) The same frustum morphed into the shape and size of the Clipbox, causing the shape of the scene it contains to morph with it.

Figure 3: The process of projecting a scene with a perspective projection, causing the view frustum to be morphed into the shape and size of the Clipbox.[1]



(a) A perspective projection frustum seen from above, and a cube moving through it towards the camera.



(b) From the perspective of the camera, the box has either moved towards the camera, or the camera has moved towards the box.

Figure 2: Moving a camera forward through a scene

[1]Image credit: opengl-tutorial.org

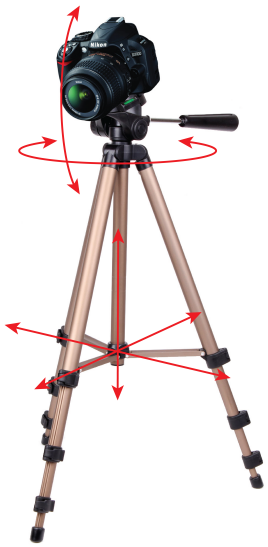


Figure 4: A camera on a tripod, with indications showing which motions the camera implemented in this task should support. The origins of the rotational and translational motions have been separated for clarity. [2]

[2]Image credit: https://www.amazon.ca/Zx600b300-Carbon-Fiber-Tripod-Cameras/dp/B007HO50NO

c) **[2.5 points]** The objective of this task is to implement a camera which behaves like one standing on a tripod, as shown in Figure 4. It should be possible to move the tripod along each major axis, as well as being able to rotate it horizontally and vertically. Each of these motions has been illustrated in greater detail in Figure 5.

For each of these motions, do the following:

(a) Create a variable which can store the value of the motion. That is, the current rotation of the camera along a certain axis, or its x, y, or z coordinates (one float or double per (rotation) axis).

(b) Add a key press handler for controlling the motion. You will need two keys every time; one to control the forward direction of the motion, and one for the reverse.

i. The key press handler should modify the variable corresponding to the rotation/translation axis you created previously.

ii. Gloom-rs by default already contains a sample key press handler you can use (it increments and decrements an arbitrary number).

iii. Keys are identified with keycodes, the full list can be found here `https://docs.rs/glutin/0.24.1/glutin/event/enum.VirtualKeyCode.html`

iv. Creating a key handler is nothing more than copying and pasting the existing one in main.rs, changing the key code, and modifying the body of the if statement to change what happens when the key is pressed.

v. Be aware that key states are checked 60 times per second (once per frame). Make sure to increment or decrement the value of your variables by something multiplied by `delta_time` to get movement independent of framerate..

vi. You are free to define yourself which keys are bound to which camera motion. Please list the keybinds you use in the report. I suggest using WASDEQ for movement and the arrow keys for rotation.

(c) Every frame, generate a transformation matrix which applies the effect of the motion on your scene.

i. Make sure you start with a "fresh" identity matrix each frame.

ii. See Figure 5 for a visual demonstrations what each of the different camera movements entail.

iii. Each camera motion only requires a single "basic" affine transformation (translation, rotation, scaling, etc).

(d) Apply each individual transformation on the transformation matrix which is later sent into the Vertex Shader.

i. Each movement results in one transformation matrix. Multiply these 5 matrices and the projection matrix together *in the correct order* to mimic the behaviour of a camera mounted on a tripod.

Some tips:

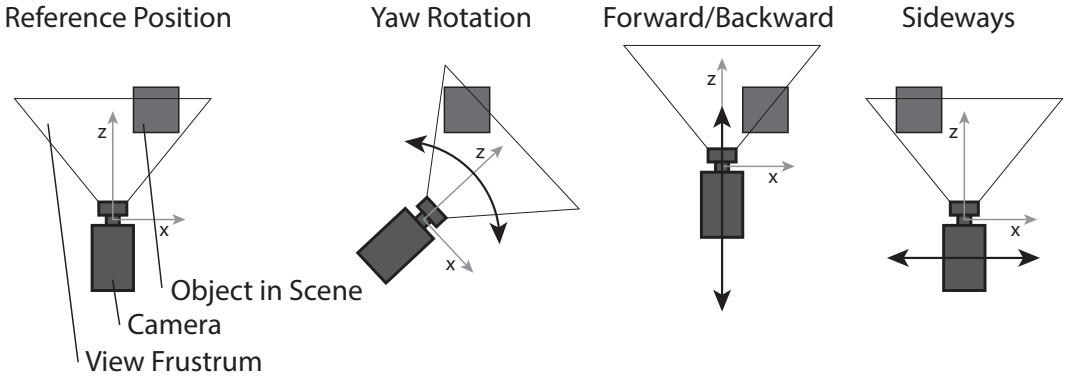
(a) Start with the identity matrix each frame. Build up the entire transformation from scratch again.

(b) In GLM, it is possible to do matrix multiplication by simply multiplying two variables containing matrices (e.g. `matrix1 * matrix2`). Don't forget that matrix multiplication is *not* commutative.

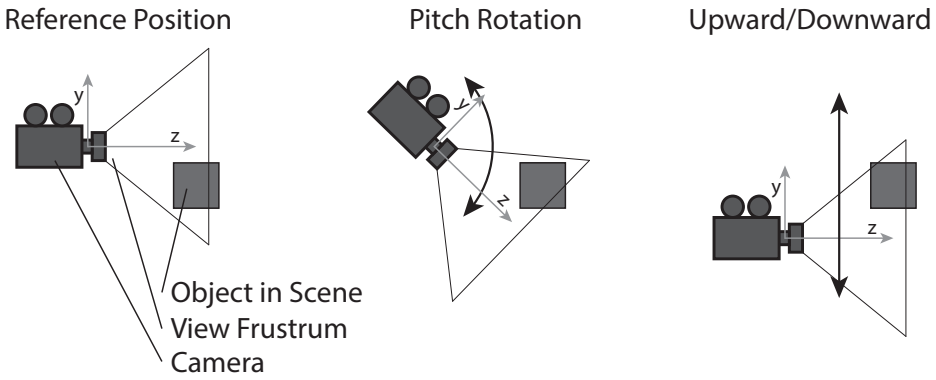
(c) Understand which origin you are rotating around and what can be seen by the camera (again, the i3t tool can help a lot here making sense of what is going on!).

(d) Try moving your camera into various situations to make sure you have something that *is* correct, and doesn't just look correct when looking straight ahead.

(e) Try to keep in mind exactly what kinds of motions you need to do in order to move the world in such a way that it appears as though the camera is looking straight ahead from the origin. There are quite a few solutions that seem to be correct at first glance that most certainly are not!



(a) A part of the camera motions you should implement in this task, as seen from above. A scene (a grey box in this case) is shown to indicate how the scene geometry moves relative to the camera's coordinate system. Try to see which specific transformations you need to accomplish the various movements, given the indicated camera's coordinate system.



(b) The remaining camera motions you should implement, in this case seen from the side.

Figure 5: A visual demonstration of the camera movements you should implement in this task. In each case, there is a "base" state shown on the left hand side. To the right of it are shown different camera movements where the camera has been displaced along the direction of the motion. You should in particular look closely at how the scene (grey box) is transformed relative to the camera's coordinate system, and use this to deduce the affine transformation needed to accomplish the particular movement.

Task 5: Optional Bonus Challenges [at most 0.3 points]

This task is optional. These questions are meant as further challenges or to highlight things you may find interesting. Their intent is to be more difficult than the tasks in this assignment, however, for those of you interested in learning more they also represent an opportunity to ensure your assignment grade is as good as possible.

As was the case with the previous assignment, getting full score on all tasks, including this one, does not give you a score that's higher than the maximum score of 5%. Similarly, doing this task is not necessary to obtain a full score on this assignment. Finally, completing all bonus challenges will at most give you 0.3 bonus points (although we'd be extremely impressed!).

a) **[0.2 points]** The camera you implemented in the previous task works well in many situations, however, it works a bit counterintuitively when the camera moves strictly along major axes without taking into account which direction it's facing.

Change the camera control scheme such that the "forward", "backward", "left", "right", "up", and "down" motions move relative to the direction the camera is facing, rather than having all of them move you in the direction of each major axis regardless of camera orientation.

Tip: You may want to limit range of motion of the vertical rotation of the camera (pitch) to between "straight down" and "straight up" to avoid weird camera movement.

Also, there's a really neat trick here if you properly understand your transformation matrices that allows you to turn movement relative to the camera into movement relative to the world, which solves this question in very few lines of code.

b) **[0.1 points]** **[report]** When passing values from the vertex shader to the fragment shader, there is a slight problem: a triangle described by 3 vertices can cover any number of pixels on the screen. As such there is no one-on-one correspondence between the outputs of a vertex shader, and inputs handed to all the instances of the fragment shader for each pixel being drawn on screen.

To solve this, output values of the vertex shader are interpolated depending on the location of the fragment and the vertices of the triangle (or primitive). You saw the result of this in task 1 of this assignment.

The way these values are interpolated (if at all), can be changed by adding special qualifiers to Fragment Shader inputs. You simply add them as a special attribute to your input variable definitions. For instance:

```
in layout(location=0) flat vec4 vertexColour;
```

GLSL supports the following interpolation qualifiers:

flat
: No interpolation is performed. Instead one vertex in the triangle is designated to be the "Provoking" vertex, from which all values are copied directly.

noperspective
: Regular interpolation takes perspective into account in order to correctly interpolate values. Enabling this interpolation type instead causes values to be interpolated in window/screen space, giving some visually oddities.

smooth
: If no interpolation qualifier is present, this is the default interpolation mode. Performs "perspective corrected" interpolation.

Create a scene (or simply a camera position) that looks different with noperspective compared to smooth, and attach screenshots of the two versions.

Tip: You might want to change the fragment shader in such a way that you get a sharp border of color within a triangle in order to more easily see the effect of the qualifier.

c) **[0.3 points]** Special effects such as smoke or fire are commonly implemented as so-called "particle systems". For large productions, particle systems are often highly advanced and configurable. However, in their simplest form they mostly come down to a lot of small geometry (a few triangles or even single points) being animated along some common motion.

One method for rendering particles is the so-called "billboard". Billboards are usually rectangles (two triangles), which are transformed in such a way that they always face the camera.

One easy way of accomplishing this is to set all components related to rotation in the transformation matrix to 0 (thus orienting the drawing geometry towards the camera), then drawing your billboard with it.

d) **[0.1 points]** **[report]** Affine transformations can accomplish many different transformations. However, when applying the same transformation on something like a 3D model, there are limits too.

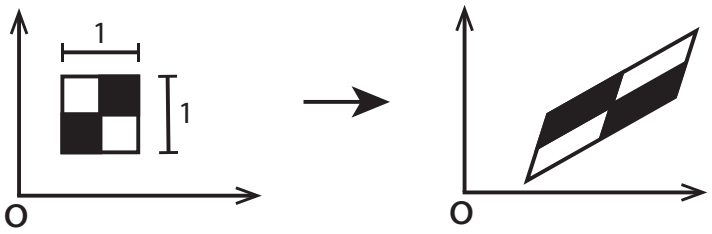
The figures in this question show pairs of geometric structures. In each pair, one or more objects are shown, as well as the same object(s) having undergone some transformation.

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

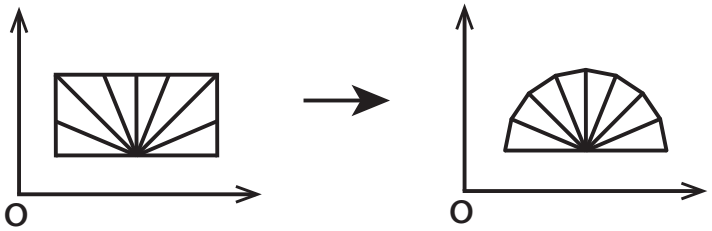
All coordinates of the shown shapes lie on the xy-plane (all z-coordinates are 0).

Additional mindbender: do the above task again, but you're now allowed to place all vertices at arbitrary z-coordinates. Which of the shown transformations are now possible?

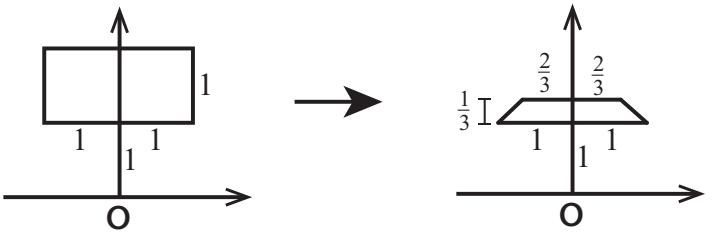
i)



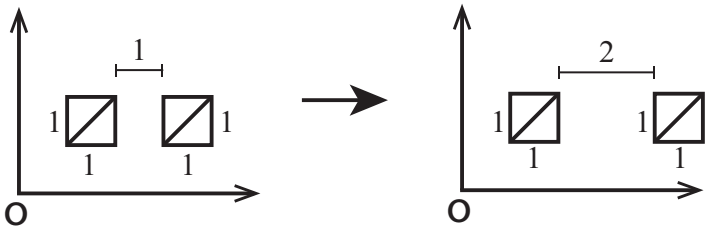
ii)



iii)



iv)



v)