Lecture 03

You're getting old, Matrix

12 Prairial, Year CCXXX

Song of the day: ***Juban District*** *by Ginger Root (2021).*

Sections

Part 1: *The XYZs of Animation*

How do we simulate movement through space? Physics teaches to do so via methods called vectors, whereby each vector contains a X-, Y-, and Z-coordinate in **cartesian space**:

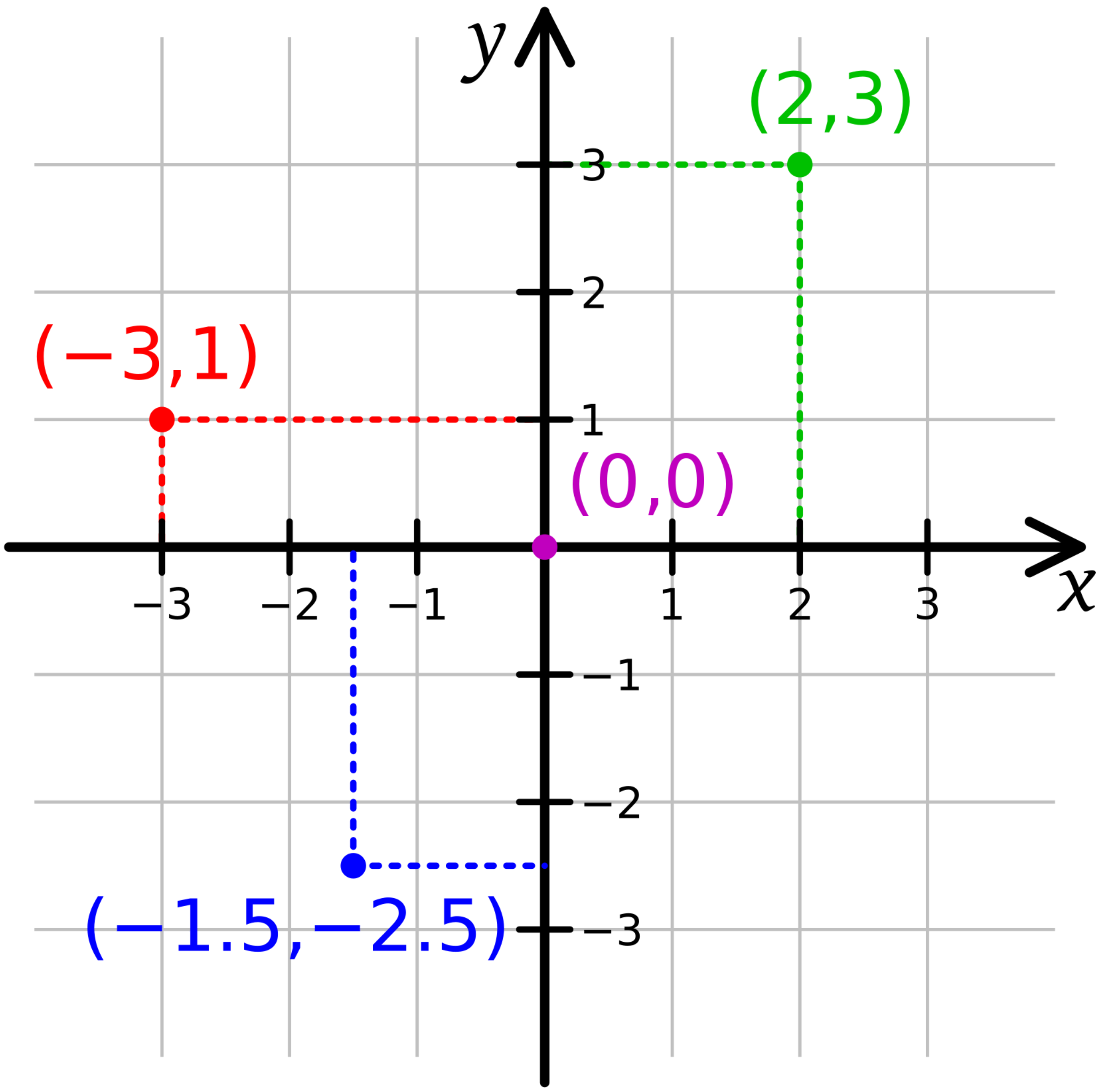


Figure 1: Illustration of a Cartesian coordinate plane. Four points are marked and labeled with their coordinates: (2, 3) in green, (−3, 1) in red, (−1.5, −2.5) in blue, and the origin (0, 0) in purple.

This makes sense—we exist as we know it in three-dimensional space, and as such it takes three values to determine a location in each of those three dimensions.

It will not surprise you to learn that modelling in video games works in exactly the same way. In just about every situation, we will be dealing with an initial set of coordinates, represented by an X-value, a Y-value, and a Z-value, and then some sort of **transformation** being done upon this set of coordinates. The resulting set of coordinates will then be displayed on our screen and appear to us as *movement*.

Naturally, there are ways with which OpenGL performs these operations, and they are based on the principles of **linear algebra** or, more specifically, on **matrix** operations:

$$\begin{array}{c|cccc|}
 & 1 & 2 & \dots & n \\
1 & a_{11} & a_{12} & \dots & a_{1n} \\
2 & a_{21} & a_{22} & \dots & a_{2n} \\
3 & a_{31} & a_{32} & \dots & a_{3n} \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
m & a_{m1} & a_{m2} & \dots & a_{mn}
\end{array}$$

Figure 2: An *m* × *n* matrix. The *m* rows are horizontal and the *n* columns are vertical. Each element of a matrix is often denoted by a variable with two subscripts. For example, $a_{2,1}$ represents the element at the second row and first column of the matrix.

Part 2: *Matrix Arithmetic*

Before we move, it may be necessary to brush up on some linear algebra—specifically, on matrix operations.

Scalar Multiplication

This one is easy. A **scalar** value is any value that has a magnitude but not a physical direction. Examples of these are temperature, ages, and mass. Vectors, by definition, have a physical direction, but because our second operand is a scalar, we don't have to worry about changing it. We simply multiply all of the values inside the matrix by that scalar value to get our result. For example:

$$4 \times \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 \times 4 \\ 2 \times 4 \\ 3 \times 4 \end{bmatrix} = \begin{bmatrix} 4 \\ 8 \\ 12 \end{bmatrix}$$

Figure 3: The matrix (1, 2, 3) being multiplied by the scalar value 4.

Matrix Multiplication

Matrix multiplication has some interesting rules that we need to be aware of. Namely:

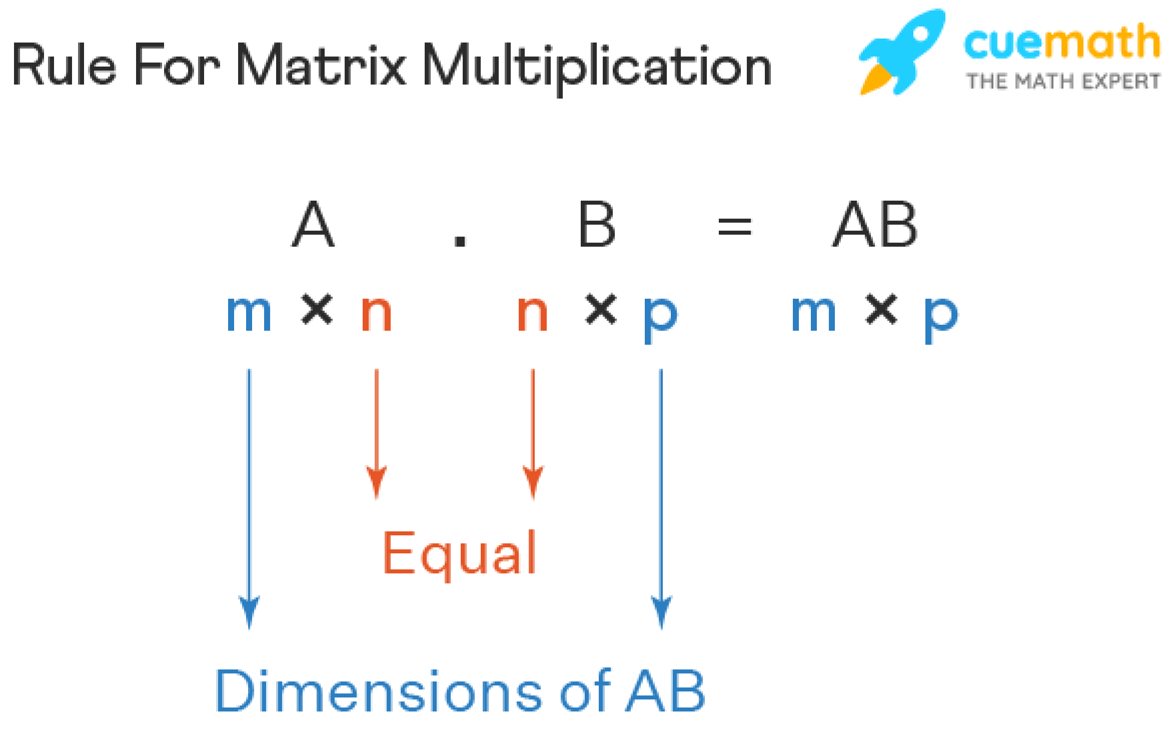
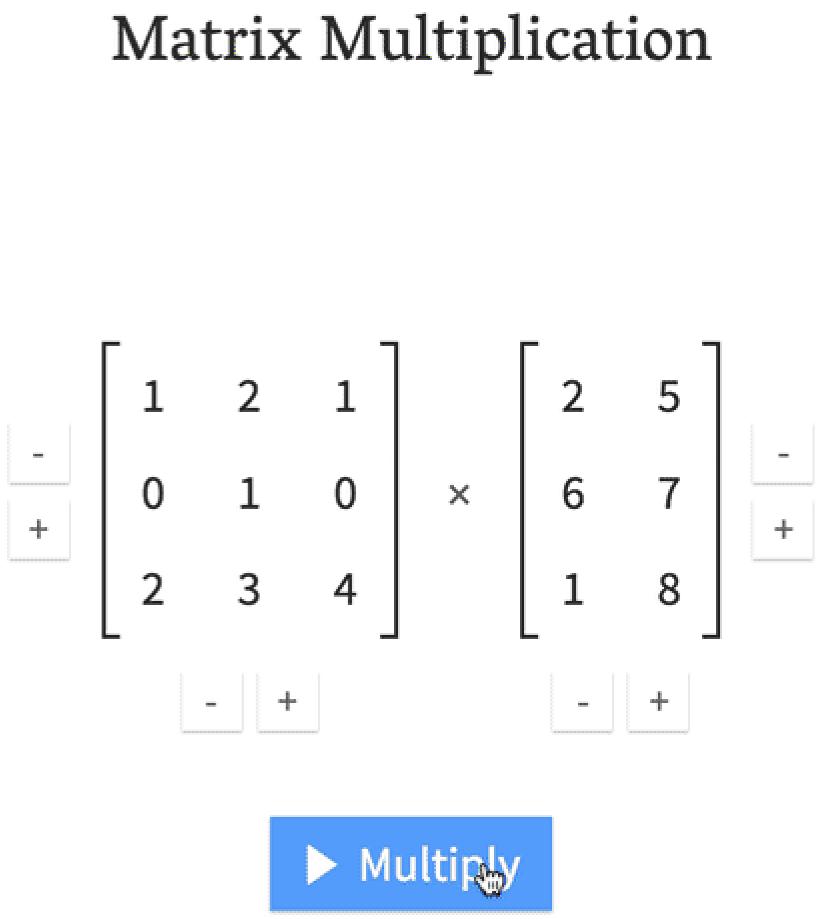


Figure 4: Matrix multiplication rules (**source**).

Essentially, the number of columns in matrix A *must* be equal to the number of rows in matrix B. For example, multiplying a 4 × 3 matrix by a 3 × 4 matrix is valid and it gives a matrix of order 4 × 4, whereas multiplication of a 4 × 3 matrix and 2 × 3 matrix would not be possible. Here's a sample animation of how it works:



In other words, we take the transpose of the second matrix, multiply and add elements together to get the result.

Figure 5: Visualisation of matrix multiplication (**source**).

Taking the 15 in location (1, 1) of the the resulting matrix as an example, we:

1. Perform 1 ($\mathbf{A}_{1,\ 1}$) * 2 ($\mathbf{B}^{T}_{2,\ 1}$) = 2
2. Perform 2 ($\mathbf{A}_{1,\ 2}$) * 6 ($\mathbf{B}^{T}_{2,\ 2}$) = 12
3. Perform 1 ($\mathbf{A}_{1,\ 3}$) * 1 ($\mathbf{B}^{T}_{2,\ 3}$) = 1
4. Add them all together to get 2 + 12 + 1 = **15**
5. Locate 16 in location (1, 1) of the resulting matrix.

Part 3: *Modeling Operations*

Why do we need matrices, of all things? It turns out that both their representative and operative abilities—that is, what they can represent and how they can interact with each other—is an accurate and elegant way of representing transformations on objects.

Scaling

For example, consider the top of a pyramid that is, say, 100 metres tall. Using cartesian coordinates, we might represent its apex as point (0, 0, 100.0)—that is, it shoots straight upwards 100 metres in the air, but does not move away from the origin in neither the x- nor the y-directions. The matrix equivalent of this cartesian notation is a **vector**, and looks something like this:

$$\begin{bmatrix} 0.0 \\ 0.0 \\ 100.0 \end{bmatrix}$$

Figure 6: A 1 × 3 matrix, or a *vector*, representing the cartesian point (0.0, 0.0, 100.0).

Why is this helpful? Let's say we wanted to make our pyramid twice as big. This would mean that its highest point—its apex—would no longer rest at point (0.0, 0.0, 100.0), but rather at point (0.0, 0.0, 200.0). This is known as **scaling**, and is the simplest of the three main transformations that we will learn how to do in this course. This is because, if you have a matrix indicating a point in space, and you want to scale it, all you have to do is multiply each of its coordinate values by that scalar value (e.g. 2). Let's look at the other two.

$$2 \times \begin{bmatrix} 0.0 \\ 0.0 \\ 100.0 \end{bmatrix} = \begin{bmatrix} 2 \times 0.0 \\ 2 \times 0.0 \\ 2 \times 100.0 \end{bmatrix} = \begin{bmatrix} 0.0 \\ 0.0 \\ 200.0 \end{bmatrix}$$

Figure 7: An example of scalar multiplication.

Rotation

Here's where matrices really come in handy. Scaling things is easy because you are multiplying a vector (something with both a magnitude and a direction) by a **scalar value** (something that only has a magnitude). But what happens when you want to operate on a vector in a certain direction?

The act of rotation—moving a certain object around a given axis—is, by definition, a directed operation, so multiplying by a scalar won't do. For this, we need a special matrix:

$$\boldsymbol{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Figure 8: The **rotation matrix**, *R*.

Here, our angle, *θ*, is the angle at which our model is being rotated in respect to the positive x-axis about the origin of a two-dimensional Cartesian coordinate system (that nightmare of a sentence will make a lot more sense once we get to programming lol). Performing a general multiplication using this matrix, we would get something like:

$$\boldsymbol{R}\mathbf{v} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix}.$$

Figure 9: A column vector **v** being multiplied by the rotation matrix ***R***.

And it's basically the same for every 2-dimensional column vector. Say we had a point in location (0, 5), and we wanted to rotate it 45 degrees:

$$\begin{bmatrix} \cos(45.0°) & -\sin(45.0°) \\ \sin(45.0°) & \cos(45.0°) \end{bmatrix} \begin{bmatrix} 0.0 \\ 5.0 \end{bmatrix} = \begin{bmatrix} 0.0\cos(45.0°) - 5.0\sin(45.0°) \\ 0.0\sin(45.0°) + 5.0\cos(45.0°) \end{bmatrix} = \begin{bmatrix} -3.54 \\ 3.54 \end{bmatrix}$$

Figure 10: An example of rotating a point.

Translation

Translation is essentially moving our points in any of the 3 cardinal directions.

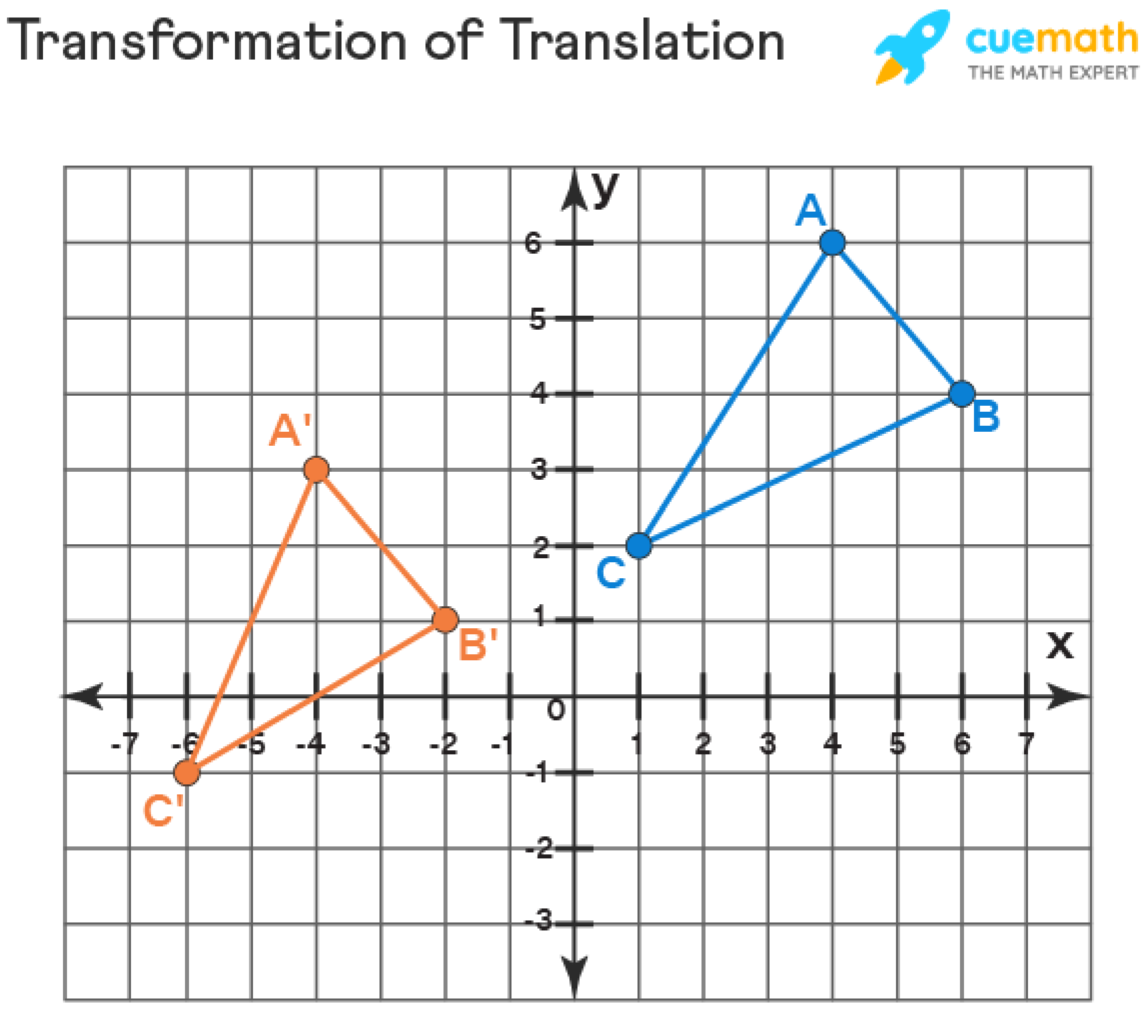


Figure 11: An example of a triangle being translated along a 2-dimensional cartesian plane.

Translations, programming-wise, are similar in "complexity" to rotations, i.e., we are multiplying a matrix by another matrix. Visualising the actual operation, though requires a little bit of change of gears. In order to perform a translation on a column vector containing an x-, y-, and z-coordinate, we need to use something called **homogeneous coordinates**.

Without getting too much into the history of homogeneous coordinates, for our purposes, using them involves simply adding an extra row in our column vector containing a 1:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Figure 12: A 2-dimensional column vector using homogeneous coordinates.

Why we end up needing this becomes clear when we look at the translation matrix that we will be using:

$$\begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 12: Transformation matrix used for translating in the x and y directions.

Let's see an example in action. If we have a point, say (4, -2), and we want to translate it two units up and one unit to the right:

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 \\ -2 \\ 1 \end{bmatrix} = \begin{bmatrix} 5 \\ 0 \\ 1 \end{bmatrix}$$

Figure 13: A column vector being transformed 2 units up and one unit to the right.

Makes sense, I hope. **Here's** a handy matrix multiplication online tool that you may use when you want to make some quick calculations when working on your games!

Part 4: *Matrices in OpenGL*

Let's take a look at some of our code from last week and see if we can find some examples of what we've been looking at.

In our `initialise()` function, we initialised our view, model, and projection matrices using our OpenGL/SDL libraries:

```
view_matrix = glm::mat4(1.0f);  // Defines the position of the camera
model_matrix = glm::mat4(1.0f);  // Defines every transformation applied
to an object
projection_matrix = glm::ortho(-5.0f, 5.0f, -3.75f, 3.75f, -1.0f, 1.0f);
// Defines the characteristics of your camera.
```

With our newfound (or rekindled) linear algebra knowledge, let's take a look at what each of their initial values actually mean. Both `view_matrix` and `model_matrix` are initialised to the same value: `glm::mat4(1.0f)`. What might this mean? If we take each element from left to right:

- **`glm::`**: This signifies that we are "extracting" something from the `glm` namespace (just like how we get `cout` from the `std` namespace).
- **`mat4`**: The is the name of the class of the object we are instantiating. As you probably guessed, this is just short of a 4 × 4 matrix.
- **`(1.0f)`**: A 4 × 4 contains 16 elements within it, so how are we initialising it with only one? The creators of these libraries did us a huge solid here by allowing us to initialise simple matrices if we don't need specific control over each of the *n* × *n* elements of a matrix. In this case, passing a `1.0f` into `mat4`'s constructor results in the creation of a 4 × 4 **identity, or unit, matrix** (*I*):

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$2 \text{ x } 2 \qquad\qquad 3 \text{ x } 3$$

Figure 14: 2 × 2 and a 3 × 3 identity matrices.

We actually saw a modified identity matrix earlier (figure 12 and 13) when translating, and we'll continue seeing them throughout the course. I wouldn't worry too much about `ortho`, since our viewpoints won't be doing too much yet, but here's a **great** article that describes just what it does, in case you're curious.

And that's really kind of it for now. Just to give you a taste of what we'll be looking at next week, I will be making our triangle scale up *ad infinitum*.

The function to scale in OpenGL also resides in the `glm` namespace, and it is called `scale`:

```
new_matrix = glm::scale(original_matrix, scale_vector);
```

Code Block 1: The syntax of a scaling call.

Here, both `original_matrix` and `new_matrix` are matrix objects of order *n* × *n*. `scale_vector` is an *n - 1* × *n - 1* `vec` object. Since basically all of the matrices that we will deal with in this class will be on the order of 4 × 4 or higher, we will be using a `vec3` object to define our scaling in the x-, y-, and z-direction. Let's see what we would do in the case of our triangle:

```
void update()
{
    // This scale vector will make the x- and y-coordinates of the
triangle
    // grow by a factor of 1% of it of its original size every frame.
    float scale_factor = 1.01;
    glm::vec3 scale_vector = glm::vec3(scale_factor, scale_factor, 1.0f);

    // We replace the previous value of the model matrix with the scaled
```

```
        // value of model matrix. This would mean that  glm::scale() returns
        // a matrix, which it does!
        model_matrix = glm::scale(model_matrix, scale_vector);
    }
```

Code Block 2: Scaling the size of our model matrix by 1% every frame.

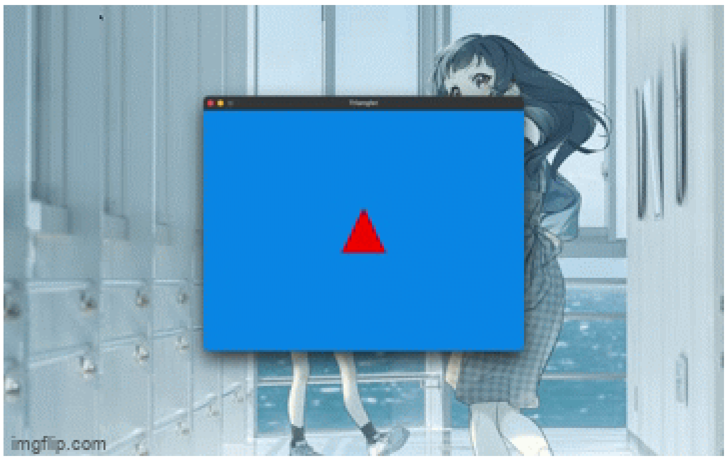


Figure 15: The result of code block 2.