# 05: Transformation and Affine Spaces

## Ressources

**Graphics Fundamentals  - Reference** and
**Shader Fundamentals - Reference**
On Canvas

WebGL(2) references
https://www.khronos.org/webgl/
https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API
https://webgl2fundamentals.org/

On Transformations and Math (Videos from lecture)
https://www.youtube.com/watch?v=kYB8IZa5AuE&feature=emb_logo

## Handin:

A ZIP file containing the following folders and files, **exactly structured like this:**

- lab05-groupXX.zip
    - lab05-groupXX
        - **All files** and folders from all exercises.

***MAC**: To compress to Zip, select a folder in the Finder, right-click and choose "compress". You have to **Zip a folder containing the folder labXX-groupXX** for this to be structured properly:*

- *[AnyFolder] ←- CHOOSE COMPRESS HERE, then name .zip appropriately*
    - *labXX-groupXX*
        - *lab folders and files*

***Windows**: Select the respective folder labXX-groupXX in the Explorer, right-click and choose Send To -> Compressed(zipped) Folder*

**Any submission not following these guidelines automatically receives 0 points.**

# Setup

Create a new folder for the lab. Download the exercise file from Canvas (Lab 05) and extract all files into it.

# 1: Completing math2d.js (50%)

Inside `lab05-exercise01-math` you will find the files `math.js` and `lab05-exercise01-math.html`.

- `math2d.js` is a math library you will have to complete
- `lab05-exercise01-math.html` is a testing environment to check if your implementations are correct.
- Complete all functions for the `V2, V3` and `M3` class. EXCEPT `determinant()` and `inverse();`

**How it works**
The 3 Classes in `math2d.js` are `V2` (2D Vectors), `V3` (3D Vectors) and `M3` (A 3x3 Matrix). They all extend `Array`, so they are Arrays themselves, but extend with custom functions specific to them.
Please read the comments of each class in the code to learn how to use them.

**Getters and Setters**
`V2` and `V3` have getters and setters for their x,y and z components. Since it is more intuitive to write `myV2.x = 5` instead of `myV2[0] = 5`. However, of course both are correct syntax. Please see the comments of the classes to learn how to use those properties.

**Matrix instance and static functions**
All functions in `V2` and `V3` were **instance functions**, but `M3` has mostly **static functions**. Since many of the `M3` functions are supposed to create new Matrices from data (eg. `translationMatrix(x,y)`, it is simpler to have them as static functions belonging to the class instead of an instance of `M3`. `translationMatrix(x,y)` is already completed as an example to show you how to use static methods here. Please read the comments of the `M3` class carefully to learn how to use it.

**`add(x,y)` versus `addV(v)` of `V2` and `V3`**
You will find several methods that look like this. Please read the documentation and parameter description of each method in the code carefully to understand the difference in use.

# 2: Transformations and use of math.js (40%)

In this exercise you will use the 2d math library to create transform-matrices for several objects. These transform- (or model-) matrices will describe the object's position, rotation and scale in space, all in one single combined matrix.
These model-matrices are then pushed to the GPU to be used in the vertex shader to position all vertices. This is much more efficient and flexible than setting the positioning by manipulating the data in the vertex position array.

## Setup

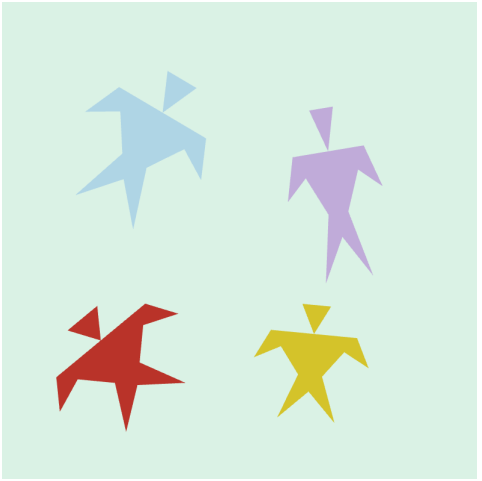
Copy your completed `math2d.js` to the other folder named `common`. Open the folder `lab05-exercise02-transformations`. Here you can see in the file `lab05-exercise02-transformations.html` that it already tries to import a file called `math2d.js` from `common`:

```
<script src="../common/math2d.js"></script>
```

You will find next to the html file a file called `modeltransform.js`. This contains not only the vertex position data and buffer setup for the model (in this case a figure), but also data for the models tint-color, position, rotation and scale on the screen as well as a matrix object. This model-matrix is a transformation matrix that we will use to make the models position, rotation and scale available to the vertex shader, to position each of its vertices accordingly.

## Goal

The goal is to get four models (for no apparent reason called Leonardo, Donatello, Michelangelo and Raphael) on the screen rendered like this:



As you can see the same figure model is rendered with different colors, position, rotation and scale on the screen, to resemble their inspirational characters.

Feel free to use other characters as inspiration :).

`modeltransform.js` is already set up to render `leonardo`, but without the use of a model-matrix, so all of `leonardo's` vertices are positioned where they would be by default.

# Complete the exercise

We know that the model matrix does not change for each of the vertices of a model. We also know it is part of the positioning, so it of course has to be a uniform in the vertex shader: `u_matrixM`.

So we need to first make sure the vertex shader can actually use a model matrix, and then assemble it in Javascript in `modeltransform.js` and set its data to the uniform `u_matrixM` when drawing.

1. Enable our vertex shader to make use of a model matrix:
    - Add model matrix uniform to the vertex shader.

```
attribute vec2 a_position;
uniform mat3 u_matrixM;
void main() {
    // TODO: use model matrix to calculate final position on the screen
    vec3 pos = u_matrixM * vec3(a_position, 1);
    gl_Position = vec4(pos, 1);
}
```

2. Assemble the model matrix:
    - Add matrix assembly from translation, rotation and scale in the `updateMatrix()` method of `modeltransform.js` using the `MultM3()`, `translationMatrix()`, `rotationMatrix()` and `scaleMatrix()` methods from your math library.
    - Make sure to follow the proper order of multiplication (see lecture slides or preparation material).
        - We want to FIRST scale, THEN rotate, THEN move. In a matrix multiplication A * B, B is applied before A (right hand side before left hand side). So when calling `M3.multM3(A,B)`, B is applied first.
    - There is lengthy help and comments in `modeltransform.js`, read them carefully.
3. Add the setting of uniform matrix data (`u_matrixM`) to the `drawModel()` method.
    - The tint color `u_tint` is also a uniform, have a look at how it's data is set, if you need help.
    - Get uniform location: `gl.getUniformLocation(...)`
    - Set uniform data: `gl.uniformMatrix3fv(...)`

Make sure to **update the model's model-matrix** (call `updateMatrix()`) before drawing, to apply any changes to position, rotation or scale!

# 2.: Simple animations (10%)

When drawing something that moves and is animated, we actually just draw a series of pictures (called frames) really fast after each other. One call of the `draw()` function executes the drawing of one frame. So when this is done, we want to call `draw()` again to draw the next frame.
Luckily there is a helpful function in Javascript, that will execute a given function once all rendering operations are complete:

```
window.requestAnimationFrame(draw);
```

In this case already with the `draw()` function as callback.

Add `requestAnimationFrame` to the END of the `draw()` function. Now `draw()` will be executed immediately after the first rendering operations are complete. And then again, and again until the end of time (or until you close the browser window).

What remains now is to update the position, rotation and/or scale of each of your models in `draw()` every frame.
For example this code, when added to the `draw()` function, will make leonardo rotate over time:

```
leonardo.rotation += 1;
leonardo.updateMatrix();
```

We have to **always update the model matrix before drawing**, to apply any changes we made to the models position, rotation or scale.

The speed of that rotation depends on how many times `draw()` is called each second, or said otherwise, how fast your computer can produce the frames: Your **Framerate**.
This code, will make leonardo slowly wiggle back and forth, **independent of your Framerate**:

```
let time = performance.now() / 1000;
let rotationOffset = Math.cos(time*5);
leonardo.rotation += rotationOffset;
leonardo.updateMatrix();
```

We take the current time (`performance.now()` is the time in milliseconds, since program start). Divided by 1000 this gives us the time in seconds. The cosine then makes the rotation oscillate.

(Only if you REALLY REALLY want to:) Can you use the logic from the RPG of exercise 02 to write a controller for one of the figures?

# 1: BONUS (10%): Determinant and Inverse

**Bonus only if you complete BOTH Determinant AND Inverse.**
Calculating the determinant of a 3x3 Matrix is doable, there are very straightforward algorithms to do so. The inverse on the other hand is rather tricky and the algorithms are fairly complex. If you want to attempt this, I recommend the method of using the adjugate of the matrix. Other methods (eg. Gaussian Elimination) are not that well suited to implement using our means.