Homework 1: Getting Started with WebGL and Transformations
EE267 Virtual Reality 2025

Due: 04/10/2025, 11:59pm

Instruction

Students should use JavaScript for this assignment, building on top of the provided starter code found on the course website. We recommend using the Chrome browser for debugging purposes (using the console and built-in debugger). Make sure hardware acceleration is turned on in the advanced settings in Chrome. Other browsers might work too, but will not be supported by the teaching staff in labs, piazza, and office hours.

Each homework will have a theoretical and programming part. The theoretical part of the homework is to be done individually, while the programming part can be worked on in groups of up to two. If you work in a group, make sure to acknowledge your team member when submitting on Gradescope. You can change your teams from assignment to assignment. Teams will share handed-out hardware (later on in the course).

Homeworks are to be submitted on Gradescope (sign-up code: **D3PYB3**). You will be asked to submit both a PDF containing all of your answers, plots, and insights in a **single PDF** as well as a zip of your code (more on this later). The code can be submitted as a group on Gradescope, but each student must submit their own PDF. Submit the PDF to the Gradescope submission titled *Homework 1: Getting Started with WebGL and Transformations* and the zip of the code to *Homework 1: Code Submission.* For grading purposes, we include placeholders for the coding questions in the PDF submission; select any page of the submission for these.

When zipping your code, make sure to zip up the directory for the current homework. For example, if zipping up your code for Homework 1, find `render.html`, go up one directory level, and then zip up the entire `homework1` directory. Your submission should only have the files that were provided to you. Do not modify the names or locations of the files in the directory in any way.

For this week's topic, course notes are available on the course website. These notes go into more detail for the mathematical concepts discussed in class. Please review the lecture slides and read the course notes before you start with this homework.
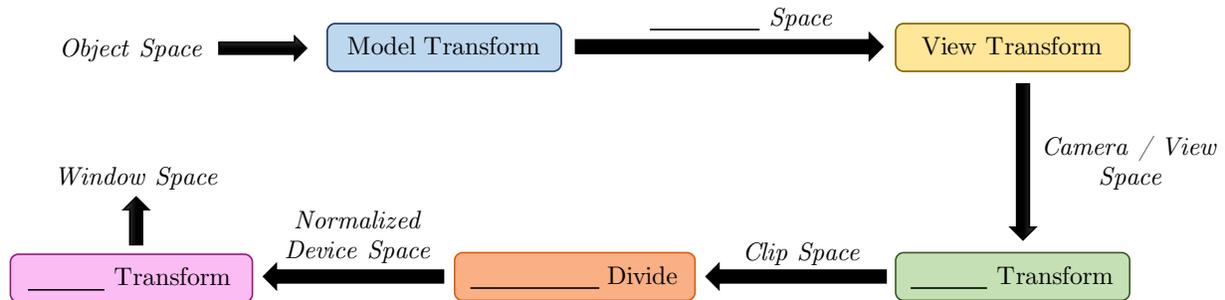
Please make sure to complete this week's lab and watch the video before you start to work on the programming part of this homework.

1 Theoretical Part

1.1 Graphics Pipeline Refresher   (*5pts*)

Before we dive into math, let's review the graphics pipeline as shown in the figure below:

Please fill in all the blanks. Make sure you review the graphics pipeline of transforming from object space to window space before starting to solve the following questions.

1.2 Understanding Transformation Matrices   (*10pts*)

(i) Show that a scaling matrix with $s_x = -1$, $s_y = 1$, $s_z = -1$ creates the same 4×4 transformation matrix as a rotation around the y axis by 180 degrees. Write out the matrices. (*5pts*)

(ii) Derive the 4×4 matrices for the two transformations below and show that they are not commutative: (*5pts*)

- (**Translation**) translation vector $\mathbf{t} = (1,\ 1,\ 1)^T$
- (**Rotation**) rotate around the x axis by 90 degrees

1.3 Graphics Pipeline Step-by-Step   (*20pts*)

In this question, we will go through the graphics pipeline step by step. Suppose we have a 3D point $\mathbf{p} = (2,\ 1,\ 3)^T$ and its normal vector $\mathbf{n} = (\frac{1}{3},\ \frac{2}{3},\ \frac{2}{3})^T$.

(i) (**Model Transform**) First, transform the point, $\mathbf{p}$, and vector, $\mathbf{n}$, into the world coordinate system, $\mathbf{p}_w$ and $\mathbf{n}_w$ respectively. Apply the following transformations in order:

- scale y by 3 and scale x and z by 2
- rotate around x-axis by 180 degrees
- translate along x, y, and z axes by 1

Report the scaling, rotation, and translation matrices, and $\mathbf{p}_w$ and $\mathbf{n}_w$. (*5pts*)

Hint: Are homogeneous coordinate representation and transformation of points and vectors the same?

(ii) (**View Transform**) Next, transform the point, $\mathbf{p}_w$, from the world coordinate system, into camera (or view) coordinates, $\mathbf{p}_v$. Use the following parameters to define the view transform:

- eye point (camera location) $\mathbf{e}$ = (0, 10, 10)
- look-at (i.e., center) point $\mathbf{c}$ = (0, 0, 0)
- up vector $\mathbf{u}$ = $(0, \frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2})^T$
- gaze vector $\mathbf{g}$ = $\mathbf{c} - \mathbf{e}$ ($\mathbf{g}$ denotes the vector pointing from point $\mathbf{e}$ to point $\mathbf{c}$)

Report $\mathbf{p}_v$. As a sanity check, apply this view transform to $\mathbf{e}$ (point), $\mathbf{u}$ (vector), and $\mathbf{g}$ (vector), and report your findings. What do you expect to see and why is this a good way to check that your view transform is implemented correctly? (*5pts*)

(iii) **(Projection Transform)** At this stage of the pipeline, the projection comes in. Apply the (symmetric) perspective projection transform to the point in camera coordinates, $\mathbf{p}_v$, using the following camera parameters:

- *aspect*: 1
- *fovy*: 90°
- *zNear(n)*: 2, *zFar(f)*: 22

Report the projection matrix, the transformed point in clip coordinates, $\mathbf{p}_{clip}$ (4×1 vector), and the transformed point in normalized device coordinates, $\mathbf{p}_{ndc}$ (3 × 1 vector), after the perspective divide. (*5pts*)

(iv) **(Viewport Transform)** The point after the perspective divide is a 3 × 1 vector $\left(x_{ndc}, y_{ndc}, z_{ndc} \right)^T$. The x and y components now have to be mapped onto a 2D screen. For simplicity, let's map the point to a small square screen with a resolution of 200×200 pixels (i.e., l_w=200 and h_w=200), where the bottom-left coordinate is (0, 0). Report the corresponding 2D window coordinate on the screen (in floating point). (*5pts*)

Programming Part PDF Deliverables

The following questions in the programming part ask you to provide written responses:

- 2.3.3 Perspective vs Orthographic (*2pts*)

Make sure to append your responses to the end of the PDF submission.

2   Programming Part

When you first open the `render.html` file in your web browser you should see three teapots, a grid, and a set of axes. The axes are defined as follows: the red line corresponds to the $+x$ axis, the green line corresponds to the $+y$ axis, and the blue line corresponds to the $+z$ axis. Note that when viewing a scene, you look down the $-z$ axis, as described in class. The starter code provides fixed model, view, and perspective matrices. In this homework you will learn about each transform individually by interacting with them via mouse movements. You might find THREE's `Matrix4()` functions useful for this assignment when creating rotation and translation matrices. You can find the documentation here. Before you begin, please update the `screenDiagonal` global variable in `render.js` with your physical screen's diagonal in inches.
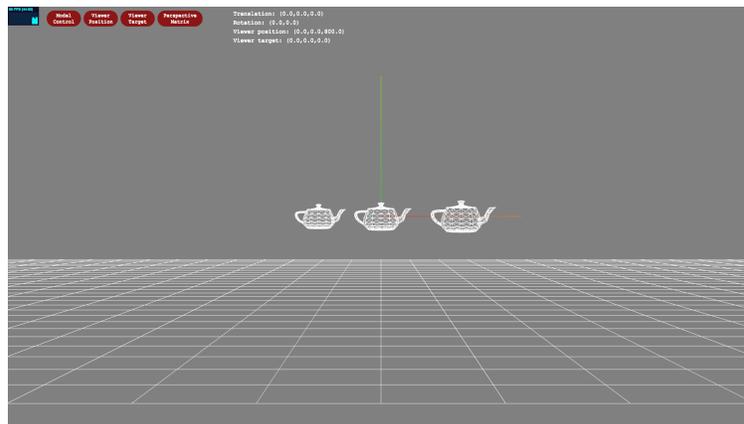
2.1   Model Transforms   (*30pts*)

When building scenes in a virtual environment, the first thing you need to do is to place different 3D objects in the scene. As mentioned in the lecture, the vertices of a 3D object are typically defined in the local space where the origin $(0, 0, 0)$ is defined with respect to the object. So if you were to load multiple 3D objects, each centered around $(0, 0, 0)$, all the 3D objects would overlap which wouldn't be very interesting. We use transformations (translation, rotation, scaling) to move the models' vertices to different positions. Typically, these transformations, defining a 3D object's position, orientation, and size in some common world coordinate system, are called the model transformation.

2.1.1   Model Translations   (*15pts*)

A fixed model transformation is already provided for you, which describes the relative position of the teapots in the scene. In this task, you will implement a simple mouse-based x, y, z translation of the three teapots. This will consist of recording mouse movements, mapping them to high level matrix parameters, and then creating the appropriate model, view, and projection matrices from these high level parameters.

2.1.1.1   Mouse Movement   First, you will have to record mouse movements. The `stateController.js` file describes a class that will handle all events (mouse, keyboard, *etc.*) throughout the course. In this class you will find the `onMove()` function which is a callback function for when mouse movement is detected. In `onMove()`, `computeMovement()` is called to compute the amount of mouse movement between frames. Implement the `computeMovement()` function to find the mouse movement. Remember to update the `previousPosition` for subsequent calls to `computeMovement()`. (*3pts*)

2.1.1.2 Mapping Mouse Movement to Matrix Parameters   It is common practice to abstract controller inputs (in this case a mouse and keyboard) from the things that end up being affected by them. We will follow this convention. Instead of the mouse movements directly affecting elements of the model matrix, we will map the mouse movements to a parametrization of the model matrix instead, and **then** construct the matrices from this parametrization.

In `stateController.js` you will find a property called `state` which keeps track of this matrix parametrization. Among them `modelTranslation` stores x, y, z translation information. We will only update this variable when the **Model Control** HTML button is engaged in your UI/browser. The control for this is already done for you in `onMove()`, but you will need to implement the mapping of mouse movements to `modelTranslation` in `updateModelParams()`. The function takes in an event as well as the per-frame mouse movement you've already implemented from above. You will want the parametrization to record the total amount of movement from the start of opening up the rendering, so keep updating the same parameter! Keep in mind that the origin of the window in jQuery (the `onMove()` function) is at the top-left of the monitor, while in WebGL/Three.js the origin is at the bottom-left.

The direction of the translations are defined as follows:

- ⇧ with $+x$ mouse movement (horizontal to the right): translate teapot along $+x$ axis
- ⇧ with $+y$ mouse movement (vertical to the bottom): translate teapot along $-y$ axis
- ctrl (or ⌘ on Macs) with $+y$ mouse movement: translate teapot along $-z$ axis

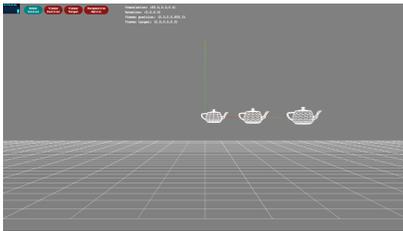To make sure your parametrization is updating correctly, check the values displayed in the browser. (*3pts*)
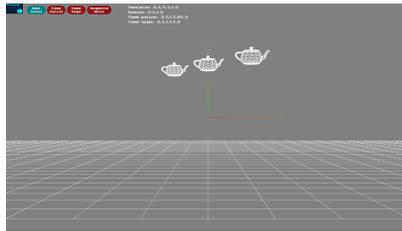

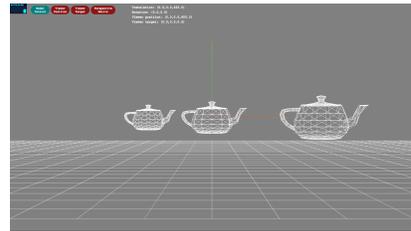**Figure 1:** *translation along +x*   **Figure 2:** *translation along +y*   **Figure 3:** *translation along +z*

2.1.1.3 Matrix Update   Now that we have the mouse movement for the model translation in `StateController.state`, we can finally form the model matrices that will render the teapots at different positions. In `transform.js`, you will find the `MVPmat` class which holds and computes model/view/projection matrices. These matrices describe transforms that are applied to each object in the scene. In the `MVPmat` class you will find the `update()` function which updates the model, view, and projection matrices every frame from the parametrization in `StateController.state`.

For this part, we're focusing on just the model matrix. Fill in the `computeModelTransform()` function to return a model matrix from `modelTranslation` in `StateController.state`, which is passed in as an argument. You may find the `THREE.Matrix4().makeTranslation()` function useful in defining the translation matrices.

Verify that your translation motion works correctly by pressing the ⇧ key and click dragging your mouse. The teapot should move in the direction of the mouse. When ctrl is pressed while dragging your mouse, the teapot should move in and out.

There is nothing you need to submit in the PDF for this task. Submit the code and we will check its functionality. (*9pts*)

2.1.2 Model Rotation   (*15pts*)

Let's add in model rotation into the model transformation. As with the translation, map mouse movements to `modelRotation` in `StateController.updateModelParams()`. This time, mouse movement with no keys pressed will administer the rotation.

The direction of the rotations are defined as follows:

- $+x$ mouse movement (horizontal to the right): rotate teapot around the $+y$ axis
- $+y$ mouse movement (vertical to the bottom): rotate teapot around the $+x$ axis
- you will not need to implement rotations around the z axis

In this task, we ask you to rotate the teapots in the following order: first around the y-axis and then around the x-axis. Make sure to use the right hand rule convention when implementing the individual rotations and consider the unit of `modelRotation` as degree.

You will form your rotation matrix in `MVPmat.computeModelTransform()` and combine it with your translation matrix to create the final model transform. The combined behavior we are looking for in this homework is for each teapot to rotate around its own origin, while the translation moves all of the teapots around together (as depicted in the video). You will need to determine the correct order in which to apply the rotations and translations, because the order will affect this behavior. Once again, no need to submit anything in the PDF for this task. We will check the functionality of your code.

Hint: When forming your rotation matrix in `MVPmat.computeModelTransform()`, you may find the THREE function `THREE.Matrix4().makeRotation*()` function useful. You might also find it useful to see the teapots from a different view point for this part. Press the space bar to toggle a top-down view of the scene.
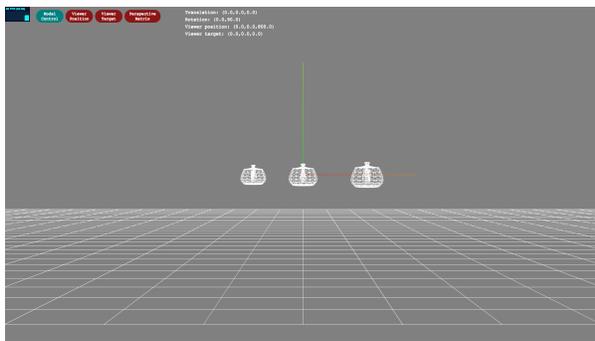

Figure 4: *model rotation around y axis, i.e. after horizontal mouse movement*

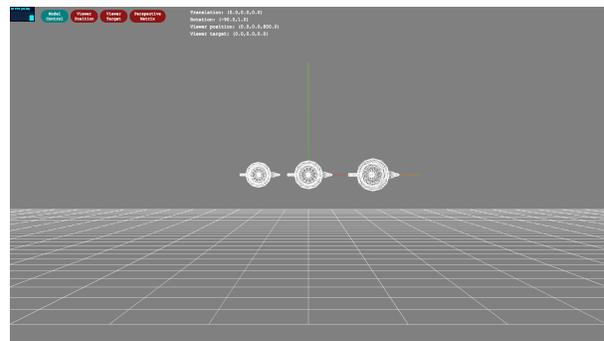**Figure 5:** *model rotation around x axis, i.e. after vertical mouse movement*

2.2 View Transform   (*20pts*)

Once the objects are positioned in world space, we apply the view transform to shift **each** model's world vertices into view space. This space will essentially dictate the position and orientation of the camera through which we see the virtual content.

2.2.1 Move Viewer Position   (*3pts*)

You will now interact with the view matrix dynamically via mouse movements. Map mouse movements to
`viewerPosition` of `StateController.state` in `StateController.updateViewPosition()`. The
updated values should be visualized in the browser. The initial value should be set to (0, 0, 800).

The direction of the position changes are defined as follows:

- $+x$ mouse movement (horizontal to the right): translate viewer position along $+x$ axis
- $+y$ mouse movement (vertical to the bottom): translate viewer position along $-y$ axis
- ctrl with $+y$ mouse movement (vertical to the bottom): translate viewer position along $-z$ axis

2.2.2 Move Viewer Target   (*3pts*)

We'll now update the point to which we are looking to, i.e. the viewer target (x, y, z). As before, map mouse move-
ments to `viewerTarget` of `StateController.state` in `StateController.updateViewTarget()`.
The direction of the viewer target changes are the same as the ones for moving the viewer position above. The initial
value should be set to (0, 0, 0). You can see the updated values in the browser.

2.2.3 Implement View Transform   (*14pts*)

A simple, fixed view matrix dictating the view transform is already provided. The fixed matrix places the camera at
800 units along the world z-axis, and points to the world origin. In this task, you will implement the view transform
function defined in class in the function `MVPmat.computeViewTransform()` in `transform.js`. From the
`state`, you may use `viewerTarget` specifying the coordinate in world space at which the camera will look and
`viewerPosition` specifying the coordinate in world space at which the camera will be positioned. Using these
two inputs, generate and output the view matrix by filling in the `MVPmat.computeViewTransform()` function.
We will check the functionality of your implementation when grading.

Attention: Three.js has a built-in function called `lookAt`. Do not use this function, because it only implements
the rotation part of the view transform (rather that translation and rotation). Just implement it yourself, otherwise
you may get unexpected results!

Hint: In this task, you assume that the viewer/camera does not rotate around z-axis. With this assumption, there
exists a unique "up" vector that works for all rotations about the other axes due to a convenient mathematical
property of how the up vector is calculated for the view transform. These two related but not necessarily identical
vectors, one that determines the orientation of the 2D rendering and one that defines the camera coordinate frame's
up direction, are often both referred to as the up vector depending on the context.

2.3 Projection Transform   (*15pts*)

Once the objects are placed in world space and then transformed to be placed in front of the camera via the view
transform, the final step is to project the 3D content onto a 2D plane so that it can be displayed as an image on a
display. This projection is called the Projection Transform. If we think of the view transform as setting the position
and orientation of the camera, we can think of the projection matrix as describing the camera's lens parameters:
focal length, field of view, and aspect ratio. Two commonly used types of projection transforms were described in
class: perspective and orthographic. You will implement each here.

2.3.1 Implement Perspective Transform   (*8pts*)

An initial fixed perspective projection transformation was given to you so that you could see anything at all in
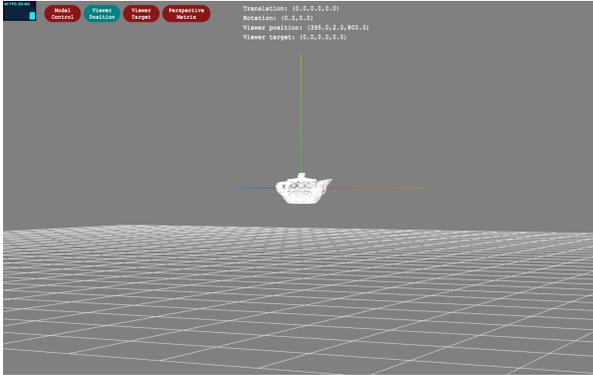the previous parts. In this task, you will fill in the function `MVPmat.computePerspectiveTransform()`

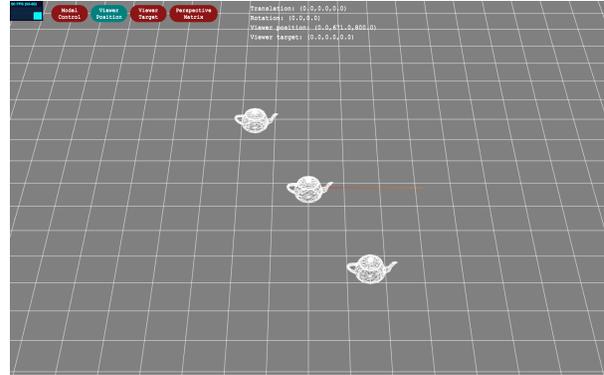**Figure 6:** *camera rotation around y axis, i.e. after horizontal mouse movement*

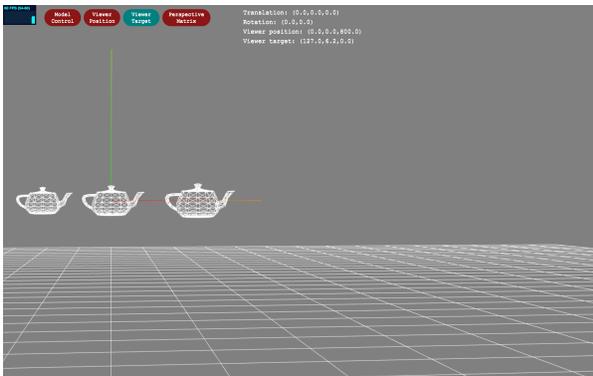**Figure 7:** *camera rotation around x axis, i.e. after vertical mouse movement*

Figure 8: *camera translation along x axis, i.e. after horizontal mouse movement*
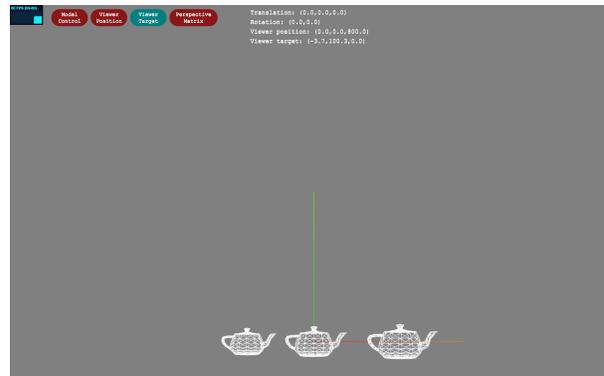

Figure 9: *camera translation along y axis, i.e. after vertical mouse movement*

that will automatically generate a perspective projection matrix based on the top/bottom, left/right, and near/far clipping planes. This follows the general definition of the perspective matrix as defined in lecture. We will need the potential for non-symmetric projection matrices for future homeworks!

To see how the `clipNear` parameter affects what we perceive, let's map it to mouse movements one last time in `StateController.updateProjectionParams()`. For the mapping, $+y$ mouse movement (vertical to the bottom) will pull the near clipping plane closer to the camera, while $-y$ mouse movements will push it farther away. In addition, to enforce the nonnegativity of `clipNear`, `clipNear` should be clamped to at least 1.

Based on the `clipNear` parameter input, implement `MVPmat.computePerspectiveMatrix()` to compute the perspective projection matrix.

2.3.2 Implement Orthographic Projection (5pts)

Now, you will implement the orthographic projection transformation, which preserves parallel lines in the perceived image. Fill in the function `computeOrthographicTransform()` based on the top/bottom, left/right, and near/far clipping planes.

2.3.3 Perspective vs Orthographic   (*2pts*)

In the PDF with your solutions to the theoretical questions, describe some of the differences that you perceive between the perspective and orthographic projections. When would you want to use the perspective projection? Can you think of some applications where you would want to use the orthographic projection?

Questions?

First, Google it! It is a good habit to use the Internet to answer your question. For 99% of all your questions, the answer is easier found online than asking us. If you cannot figure it out this way, post on piazza or come to office hours.