Assignment #2: Transformations

# Basic Info:

This assignment is graded, and it worth 25 pts (out of 500 pts in total, i.e., 5% of your grade) The deadline is April 30 (Sunday), 11:59 PM

If the syllabus and/or class notification made by instructor or TAs indicates a different deadline, please refer to the latest information.

# Repository setup:

Unzip the assignment material to your local file system (please unzip if you're reading this line inside the compression file preview now) and you can optionally use git to control your version.

You can continue with an IDE or using Chrome DevTools, but we strongly recommend using of IDE (for instruction about how to use an IDE, please refer to TA sections or A1 instructions).

If you want to debug with Chrome DevTools, this [debugger instruction](https://docs.google.com/document/d/e/2PACX-1vSi14Mb-_6qLN7wVA4-4NxqYR-3qHLy7ndjB2G0Ba6TCYHn_KGmrPbu-fCDtHkv9QcGBNqLUUdT6fu1/pub) can be helpful when your code does not work.

# Getting Started:

If you're using IDE, never mind these steps. IDE will help you with the local server and file content changing.

Open the demo exactly as you did in Assignment 1: Run a dummy web server, navigate to the URL localhost:8000, observe the initial animation we provide, open Chrome developer tools, and perform the steps to map your local file folder as a Chrome workspace.

At that point you'll be safe to edit your files without your edits disappearing or changing the wrong files. Then, proceed as follows.

# Preliminary Steps: Using the Code Library

In order to use our library, tiny-graphics.js, you the programmer must provide additional code: three custom JavaScript classes. Each of the three will be a subclass of a different base class

from tiny-graphics.js: Shape , Shader , and Scene . A defines a 3D shape to draw, a

Shape

defines some code for coloring in triangles (perhaps as though they were illuminated by

Shader

light sources), and a class tells the 3D canvas and web page what you want to happen

Scene

(usually, drawing a scene by placing shapes). The three subclasses must define certain methods

correctly for the library to work, or nothing will get drawn.

In order to get you started, we have provided examples of those custom subclasses in your code, and you only have to tweak them to do this assignment. One of the Shapes we give, Cube, is

complete, while the other, is left empty for you to fill in.

Cube\_Outline

The example scene we provided in example/transform\_sandbox.js is a fully defined working example for you. The class, called Transforms\_Sandbox, draws the extremely simple scene of boxes and balls you see when you initially run your Assignment files. It has the bare minimum to start using graphics to build your first scene. Start your coding work there.

# Experimenting with Transforms\_Sandbox

Inside the method of class Transforms\_Sandbox, you will see some lines of code that

display()

call " draw() " on a particular shape, causing one of that shape to appear in the scene. You will

also see other lines of code that modify a variable that contains a 4 by 4

model\_transform

matrix. These lines of code perform translations, rotations and scales -- the main math operations you need to understand to get started with graphics. To call them, pass in a 3x1 vector (a Vec, or a regular JavaScript array of three floats). In the case of rotation(), a scalar (representing the

angle in radians) must be also provided along with the (representing the axis of rotation).

Vec

Mat

tiny-graphics.js

Read the code comments above the work.

Vec

vec3

vec4

Vetors

and

classes in

to see how they

Note I:

and

can generate new

of dimension 3 and 4 respectively.

Note II: JavaScript has no operator overloading, so operations like etc. will not

+, -, \*, +=, \*=,

compile if you try to use them on the vector and matrix types (Vec and Mat). Instead use

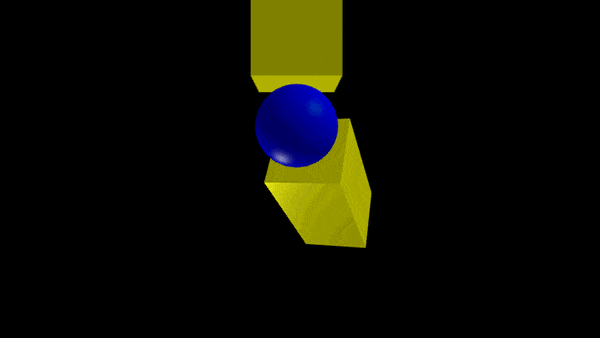
as shown in the class definition, and assign its return value back

times()

Transforms\_Sandbox

into your matrix to incrementally modify it. These can be chained together like so:

M = M.times( T ).times( R ).times( S ); this.shapes.box.draw(context, program\_state, M, this.materials.plastic.override(yellow));



Play with the code in and re-run your program until you are

example/transforms\_sandbox.js

comfortable with the effects of changing numbers passed in to the transforms. Move around the

calls to to place new shapes. Put your operations into loops to draw lots of shapes and

draw()

test your understanding of JavaScript.

Once you are comfortable with this, begin implementing the graded requirements below (a stack of cubes).

Your final scene must go in the in order to be graded.

**assignment2.js**

That class exposes its constructor to you, where you can see how it sets up the camera, lighting shader, and materials (which contain quantities related to how shapes and lights will interact), which are all defined there to help you do the actual assignment requirements.

In order to select the scene

Assignment2

are ready to switch you must change your

to display instead of Transforms\_Sandbox , once you

. Simply replace where the

main-scene.js, line:28

name with Assignment2. It will draw that instead.

Transforms\_Sandbox

We have marked places in the code with "// TODO" to demark anywhere we intend for your final code to be in order to get points. All of these places you modify are inside the file

assignment2.js .

## Graded Steps

## Up to 25 points of credit. There is no partial credit for any individual requirement.

Implement the assignment in clean and understandable code. Each required part must successfully draw and show up onscreen in order to count.

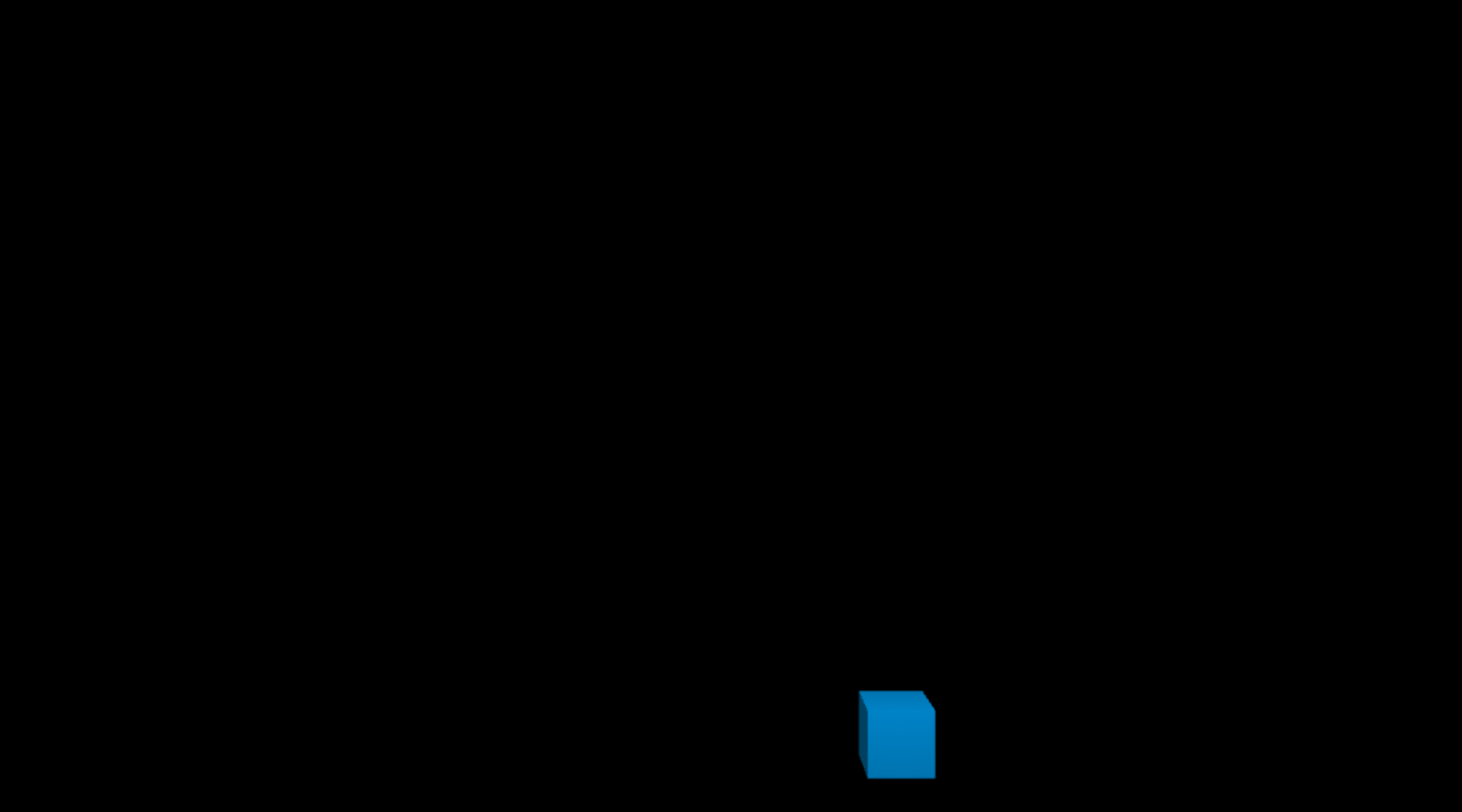
NOTE: Please use the default camera definition defined in the assignment2.js .

If any parts are unclear, ask in Discussions on BruinLearn.

## Point distribution:

1. Modify our template, which displays WebGL in an HTML canvas, without introducing errors –

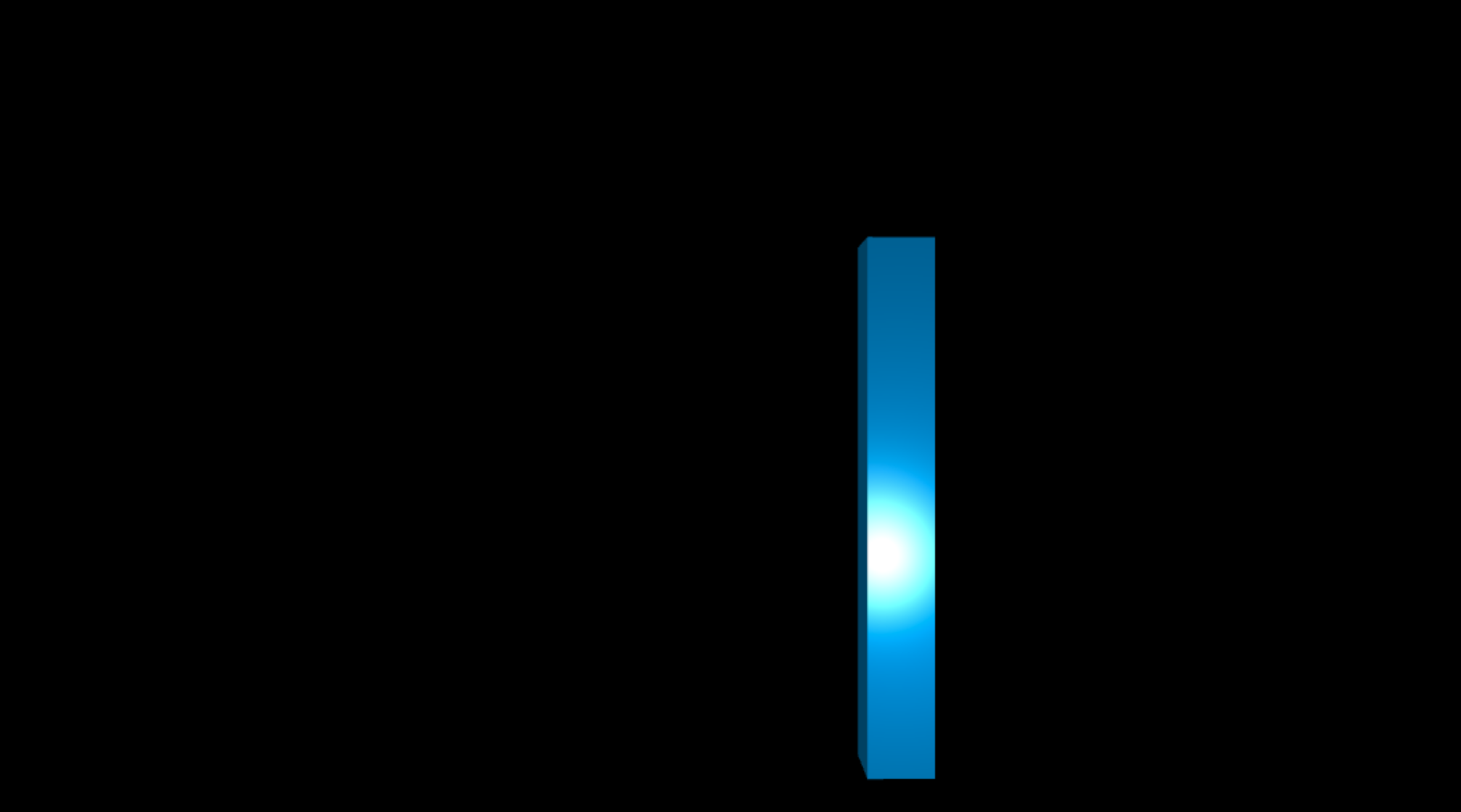
2 points.



Hint: Before you begin to make changes, you should see a single blue cube as above.

1. Display a stack of eight (8) unit cubes starting from the origin and extending upward – 5 points.

Instance each of the eight cubes from the same geometry data (we defined this as "box" in your scene's constructor function). Due to our other provided code in the constructor, the boxes will appear with a symmetric perspective projection, with square aspect ratio (not stretched or squeezed). The initial camera position at the given position (-5, 10, 30) is far back enough to view the scene. Note that to place a camera there is the opposite transform action that would be used to place a regular shape.



Hint: After this step, you should see a stack of 8 blue (shiny) cubes as above.

1. Make the stack of boxes sway like a blade of grass in the wind. It should sway back and forth once per 2 seconds. Be exact if you can.
   1. Without passing through one another, the boxes must rotate over time to a maximum angle of .05\*Math.PI (for each level of box). Place the hinge of each box's rotation motion exactly at the top left edge of the box underneath it. Each box touches the previous in exactly the same way and remains in constant contact, precisely connected only by the correct edge. The boxes may not separate from each other along this edge; no floating geometry is allowed – 4 points.

Hint: Remember that you can chain more than just one translation and rotation together when adjusting your transformation between drawing shapes.

* 1. Fluidity of your rotation motion matters. Functions of the form

f(t) = a + b\*sin(w\*t)

are useful for modeling periodic motion, where " a " is large enough that the rotation angle

does not go negative and cause boxes to collide. Use

program\_state.animation\_time

t , which is how long the program's been running in milliseconds – 2 points.

for

* 1. Iteratively place the rest of the moving box segments on top of one another to complete the swaying motion. Use a hierarchical approach - each box's transform is the "child" of the previous transform, derived from its matrix value – 1 points.

Hint: To make structures that have a parent-child hierarchy like this, you must

incrementally create your transform using to post-multiply each new term

times()

onto the right side of your matrix variable. Do this to incrementally change it from the value that drew the previous (parent) box. Never do a pre-multiply for this assignment; for certain reasons that's not as useful for designing structures that you think of as a hierarchy of shapes.

You should learn to organize your code into functions, instead of describing your whole scene in display() . For part (c) we recommend moving your code for drawing one box into

draw\_box()

display()

the blank

method, and then calling it many times from

using a for

loop. Changing the function signature (arguments) to is allowed, since it may be

draw\_box()

necessary to know which box (numbered from bottom to top) you're drawing for color purposes.

* 1. Fill in code inside your class method make\_control\_panel() to implement a button to

help us with grading. To do this, call as already shown inside the

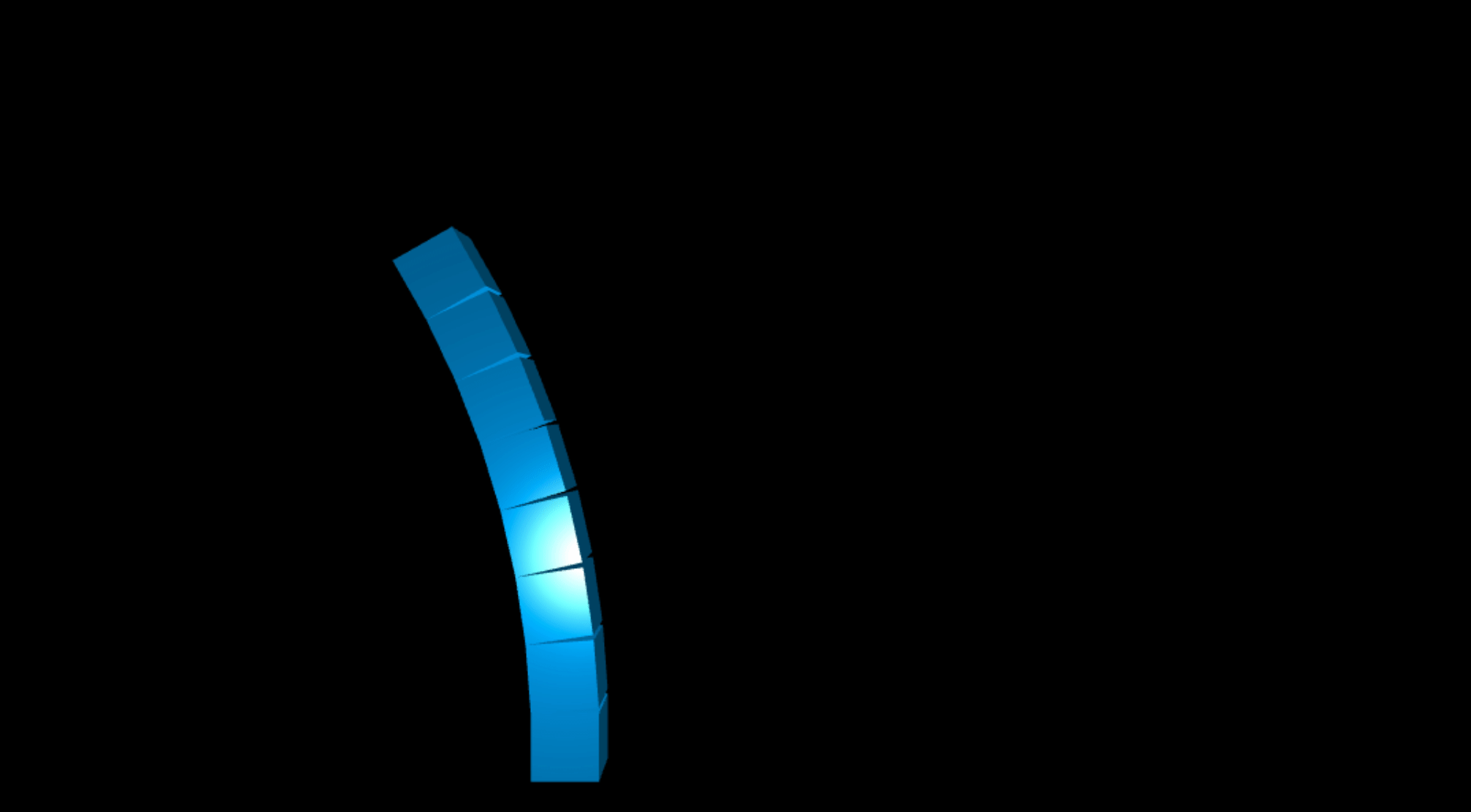
key\_triggered\_button()

method, and make sure the second argument is [ "m" ] so that we can press the m key to test your button. Fill in the third argument, the function that gets executed each button press, so as to make it toggle the swaying motion on and off. When the swaying is turned off, your

blade of grass must be extended out to the maximum possible angle of so

.05\*Math.PI

that we can see the gaps between your boxes along the left side – 1 points.

 (This is an animated GIF)

Hint: After implementing the sway back and forth, it should look like the above. In this GIF animation, the cubes are swaying back and forth (once per second) In this

GIF, we let the cube stack swaying and then pressed to set it still to the maximum angle



M

1. (a). Color each box differently (and randomly) from the one underneath, for contrast. So that colors stay the same from one frame to the next, base your colors on persistent variables

you store in your class, which means you'll use the

this

points.

Hint: One possible way to generate random color is

keyword to declare them – 3

color(Math.random(),

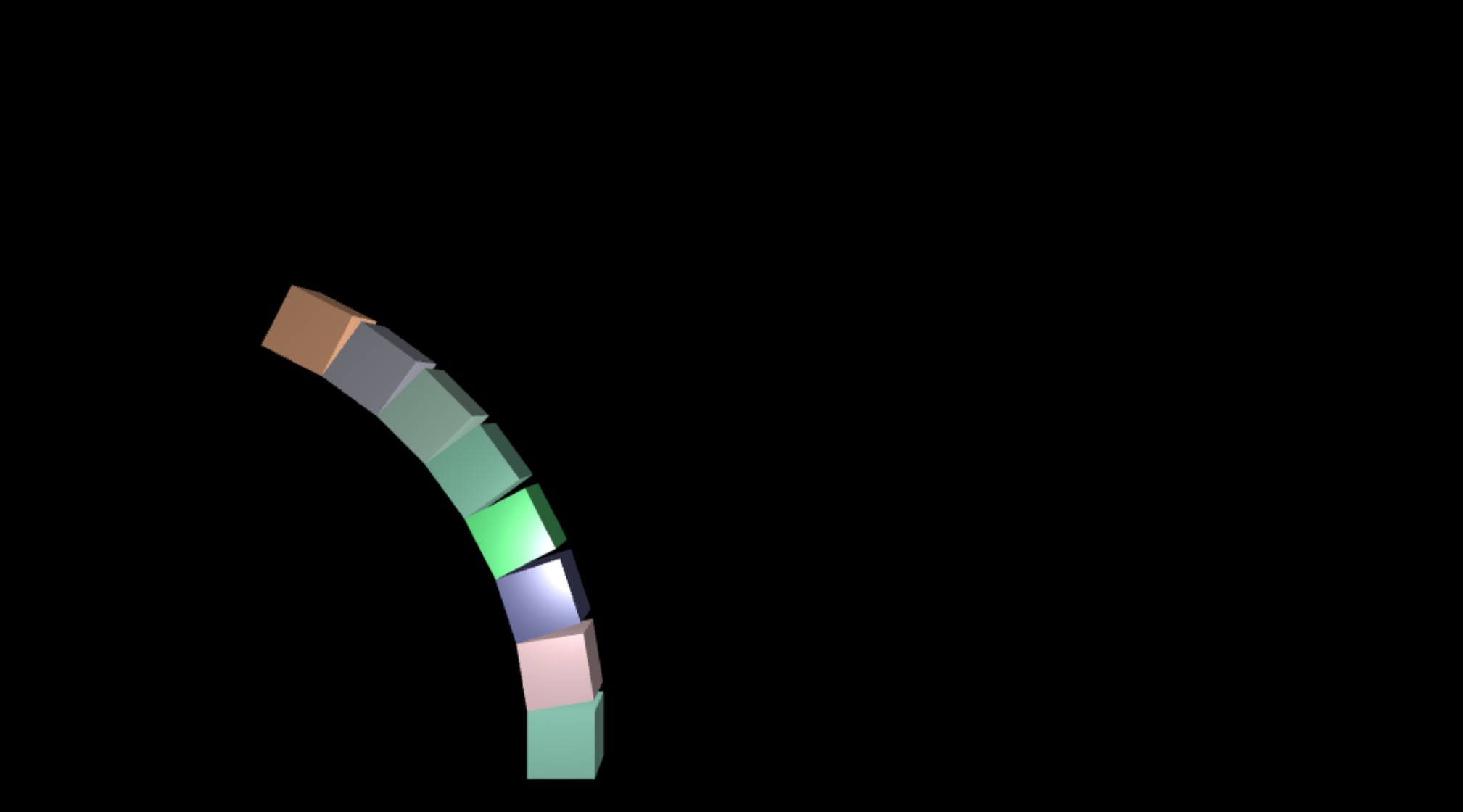
Math.random(), Math.random(), 1.0). Or you can change the range of the random numbers to ensure the color will not be too dark.

(b). Fill in the function to somehow cause your class's colors to reset to

set\_colors()

different values. Once you do this, pressing the 'c' key (which already calls set\_colors) should set the colors of the boxes to a new set of random colors – 1 points.

Shape



1. Draw each cube’s outline (the edges) in white. For this, you will need to design a new

subclass. Fill in the empty parts we left in the class, defined in your

Cube\_Outline

file. It will be like your Cube 's definition above it, except with your own

assignment2.js

brand new arrays. Each vertex gets a position and a color (color replaces in this

normal

example). Define the positions of your cube outline as x,y,z coordinates of the cube points. Order them in pairs, so that each pair is one edge of the outline. You may NOT have any

extra edges going across diagonals. Set each color value in the array to full white -

color

making sure the list has as many entries as the instead use "this.indices = false".

positions

* 1. To actually draw your outline, you will need to call

list. Do not make an indices list -

on it and pass in a material

draw()

that is compatible with it. That's because you just changed its available fields to

Shape

"positions" and "color", and so the type of we use on it has to know what to do with

Shader

that information. The variable we called "this.white" already holds such a proper,

compatible material -- so just pass that one in as your third parameter to your

draw()

outline. This should draw it using exactly the colors you made for it (white colors). Lastly, to actually draw lines instead of triangles (letting it know that there's pairs of vertices instead of

triples), you must pass in the string "LINES" as the last argument to – 3 points.

draw()

* 1. Rather than drawing your outline at all times, fill in code inside your class method

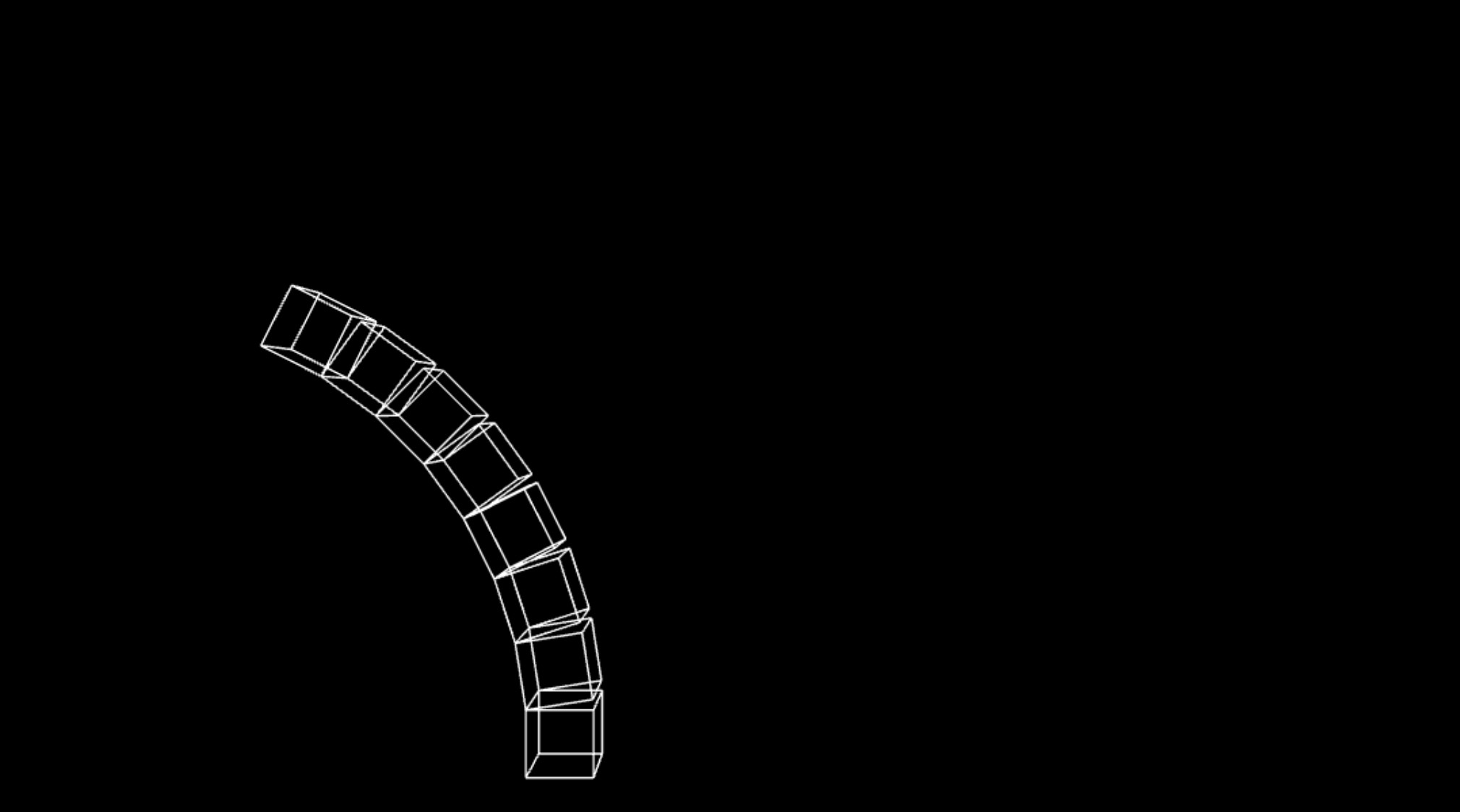
to implement another button. Call as

make\_control\_panel()

key\_triggered\_button()

shown, and make sure the second argument is ["o"] so that we can press the o key to test your button. Fill in the third argument, the function that gets executed each button press, so as to toggle a flag that you'll use to turn the outline on and off. When the outline is off, draw

the boxes normally (colors, lighting, etc.). When the outline is on, draw the outline instead on all boxes (it's OK to skip doing this for the bottom-most box since it's special, as explained below) – 1 points.



1. Triangle strips are a common graphics primitive. Implement the even number (count from bottom to top, starting with index 1, i.e., draw the 2nd, 4th, 6th, 8th box using triangle-strips method) box's geometry as a single triangle strip primitive. This is like making a cube that can unfold into a single line of triangles.

Create a new

subclass with whatever correct entries in the

and

lists that will make that happen. Give each vertex a that is simply equal to the

Shape

position

indices

normal

draw()

"TRIANGLE\_STRIP"

position. When calling

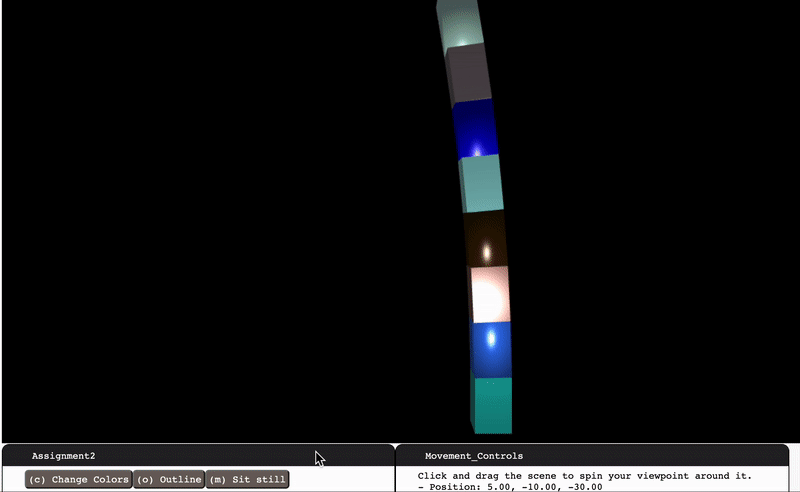
to make this box, pass in the string

for

the fifth argument so that the graphics card uses triangle strip indexing. This box should appear lit up differently than the others, especially near edges as you move the camera around it, because of its unusual layout – 1 points.

1. Scale your boxes so that instead of being unit cubes, they are stretched to 1.5x their length only along the Y axis. Adjust your translations accordingly so that the correct hinge contact points are maintained throughout the swaying motion. Again, no floating geometry is allowed. Neither are any shearing effects that deform the boxes to non-right angles during motion. Prepare your left-to-right chain of transformations accordingly for each separate shape so that those things cannot happen – 1 points.

The final reference video with mouse interaction:

(This is an animated GIF)

# Submitting Assignment 2:

Compress (zip) the whole assignment folder into a ZIP format compressed file. Submit it to the Assignment in Bruin Learn where you download this.

Please make sure you included everything in the assignment, rather than only the file(s) you made changes. The structure of the compressed file should be the same with the provided assignment material.