Computer Graphics

Lab 2: Transformations and Projections

Schedule

Lab 1	Introduction to WebGL	10.3.2017 / 14.3/2017	-
Lab 2	Transformations and Projections	21.3.2017 / 24.3.2017	Lecture: 14.3.2017
Lab 3	Scene Graphs	28.3.2017 / 31.3.2017	Lecture: 21.3.2017
Lab 4	Illumination and Shading	4.4.2017 / 7.4.2017	Lecture: 28.3.2017
Lab 5	Texturing	25.4.2017 / 28.4.2017	Lecture: 4.4.2017
Lab 6	Advanced Texture Mapping	2.5.2017 / 5.5.2017	Lecture: 26.4.2017
Lab 7a	CUDA	9.5.2017 / 12.5.2017	
Lab 7b	VTK	12.5.2017	

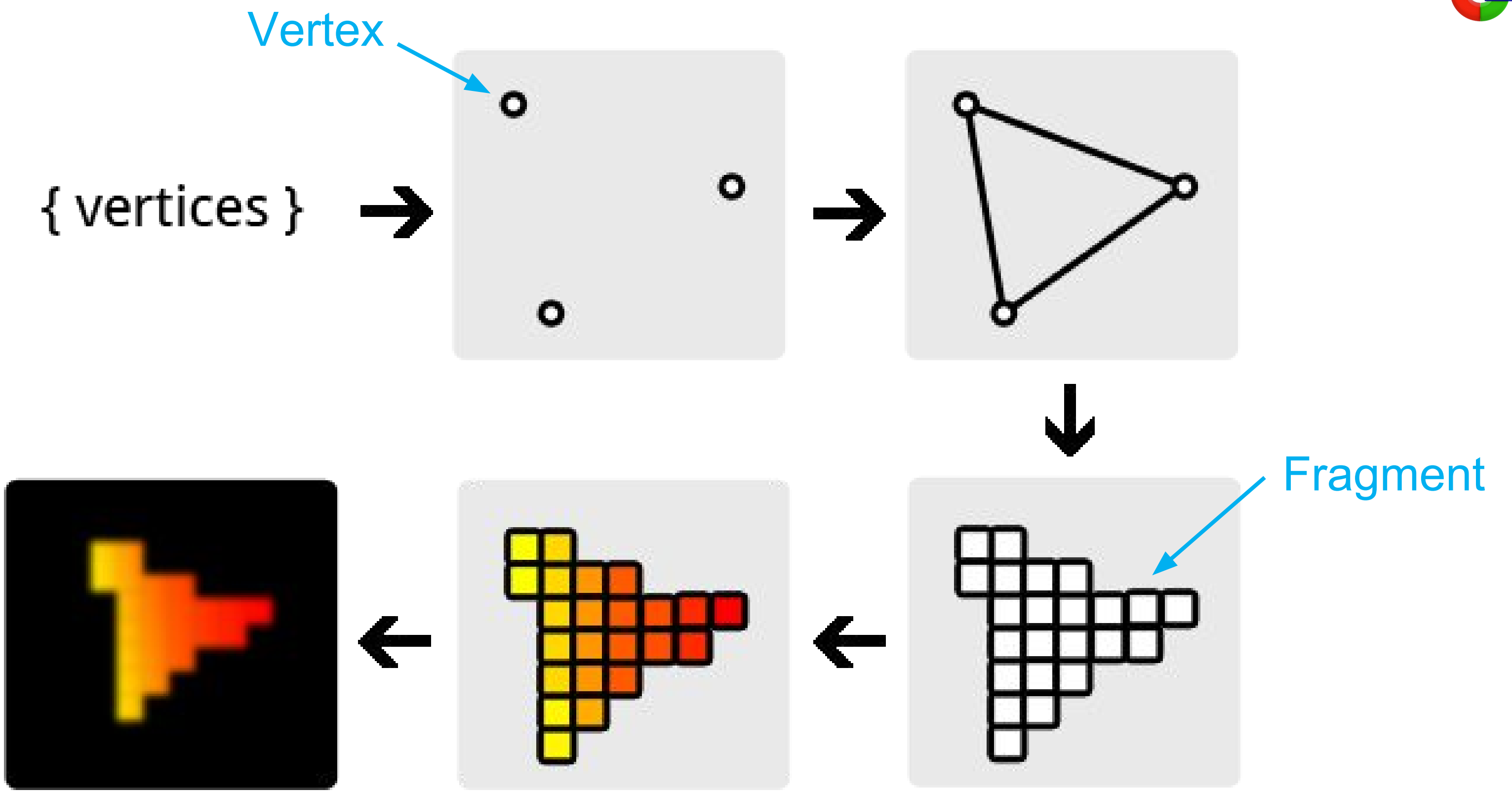
JOHANNES KEPLER
UNIVERSITY LINZ

Institute of
Computer Graphics

Slides and lab material

www.cg.jku.at/teaching/computergraphics/lab

Recap: Rendering Pipeline

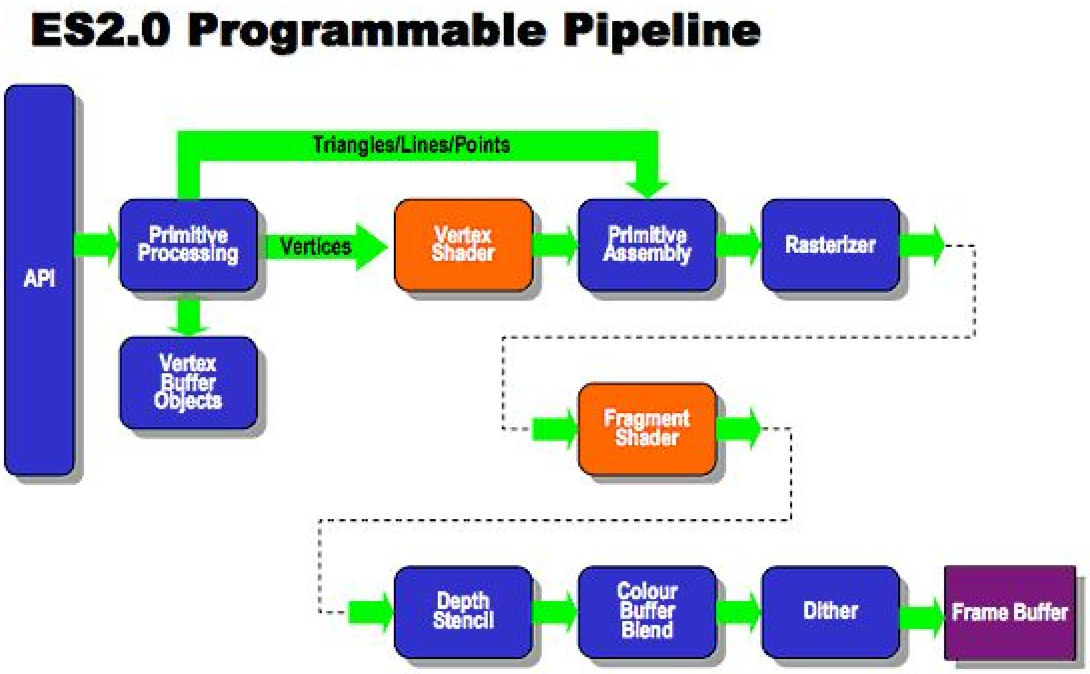


https://open.gl/drawing (adapted)

Recap: Programmable Pipeline

JOHANNES KEPLER UNIVERSITY LINZ
Institute of Computer Graphics

Buffer Objects
↓ attributes
Vertex Shader ← uniforms
↓ *gl_Position* varyings
Primitive Assembly
↓
Rasterization
↓ varyings
Fragment Shader ← uniforms
↓ *gl_FragColor*
Per-Fragment Operation
↓
Framebuffer



Fork me on GitHub

Dev Environment: Lab Package

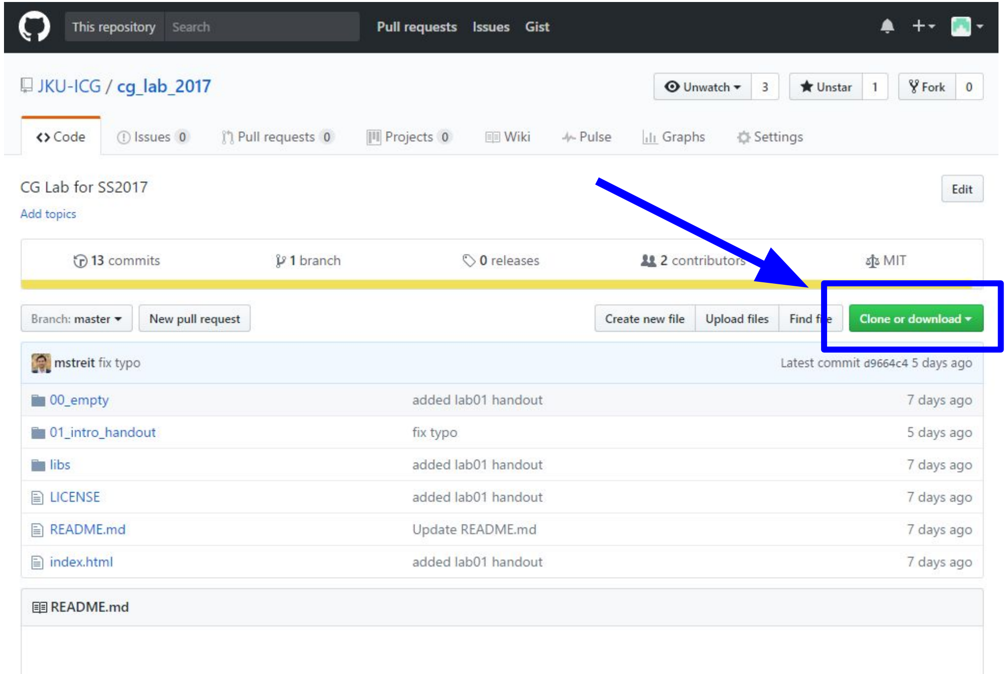
JOHANNES KEPLER UNIVERSITY LINZ
Institute of Computer Graphics

Hosted on GitHub: https://github.com/jku-icg/cg_lab_2017

The repository will be updated during the lab with the new projects.

To get started (now):

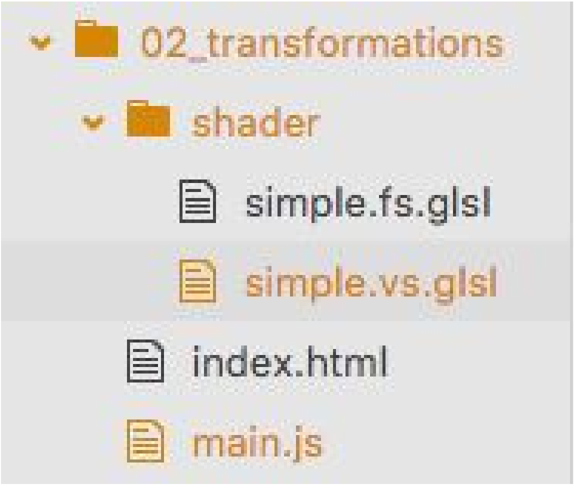
1. Download the zip
2. Extract the folder
3. Open Atom editor
4. Use “Open Folder” in Atom
5. Start server on any port
   (Plugins -> Live Server)



Dev Environment: HTML5, JS, CSS

WebGL → OpenGL in the Web-browser based on OpenGL ES 2.0

Basic project structure:



index.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Empty</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <!-- include helper library for matrix computation -->
    <script src="../libs/gl-matrix.js"></script>
    <!-- include our framework with utilities -->
    <script src="../libs/framework.js"></script>
    <!-- include the main script -->
    <script src="main.js"></script>
</body>
</html>
```

main.js

```
//the OpenGL context
var gl = null;

/**
 * initializes OpenGL context, compile shader, and load buffers
 */
function init(resources) {
  //create a GL context
  gl = createContext(400 /*width*/, 400 /*height*/);

  //TODO initialize shader, buffers, ...
}

/**
 * render one frame
 */
function render() {
  //specify the clear color
  gl.clearColor(0.9, 0.9, 0.9, 1.0);
  //clear the buffer
  gl.clear(gl.COLOR_BUFFER_BIT);

  //TODO render scene

  //request another call as soon as possible
  //requestAnimationFrame(render);
}

loadResources({
  //list of all resources that should be loaded as key: path
}).then(function (resources /*loaded resources*/) {
  init(resources);
  //render one frame
  render();
});
```

← 2. init OpenGL

← 3. render frame

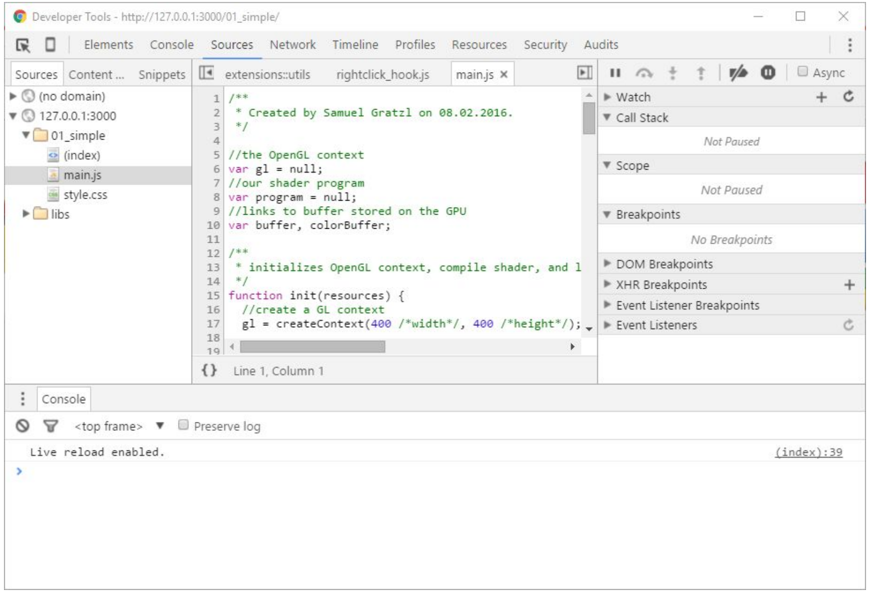
← 1. load external resources

Dev Environment: Developer Tools

Know the Web Developer Tools of your favorite browser

- Chrome, Firefox, Edge, Safara, … → usually F12

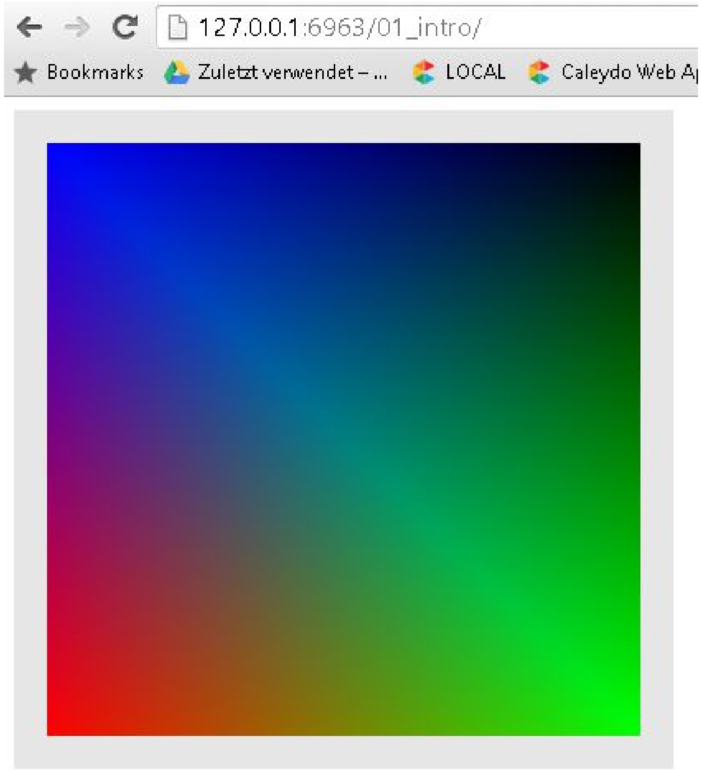
Great for debugging JavaScript code, manipulating CSS & DOM, …



Recap: Colored Triangle

First Application: Colored rectangle

a. initialize context
b. define buffer, compile shader
c. draw rect, i.e. two triangles
d. specify uniforms
e. specify color per vertex



Agenda for Today

Transformation pipeline

Model-view transformations

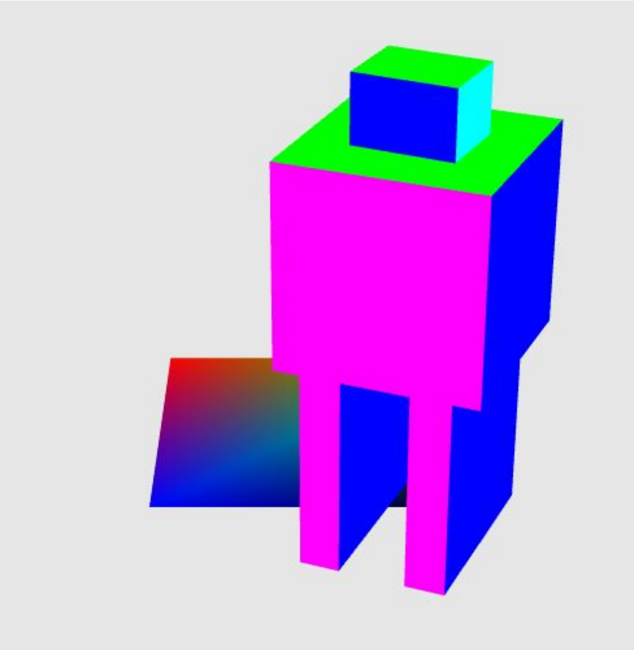
- Translate, scale, rotate, animations

Creating geometry using the index buffer

Projective transformations

- Orthographic and perspective projection

Camera transformations

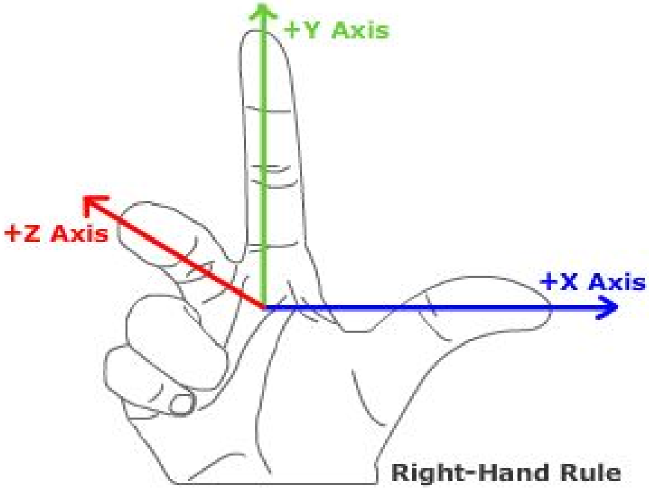


OpenGL's Coordinate System

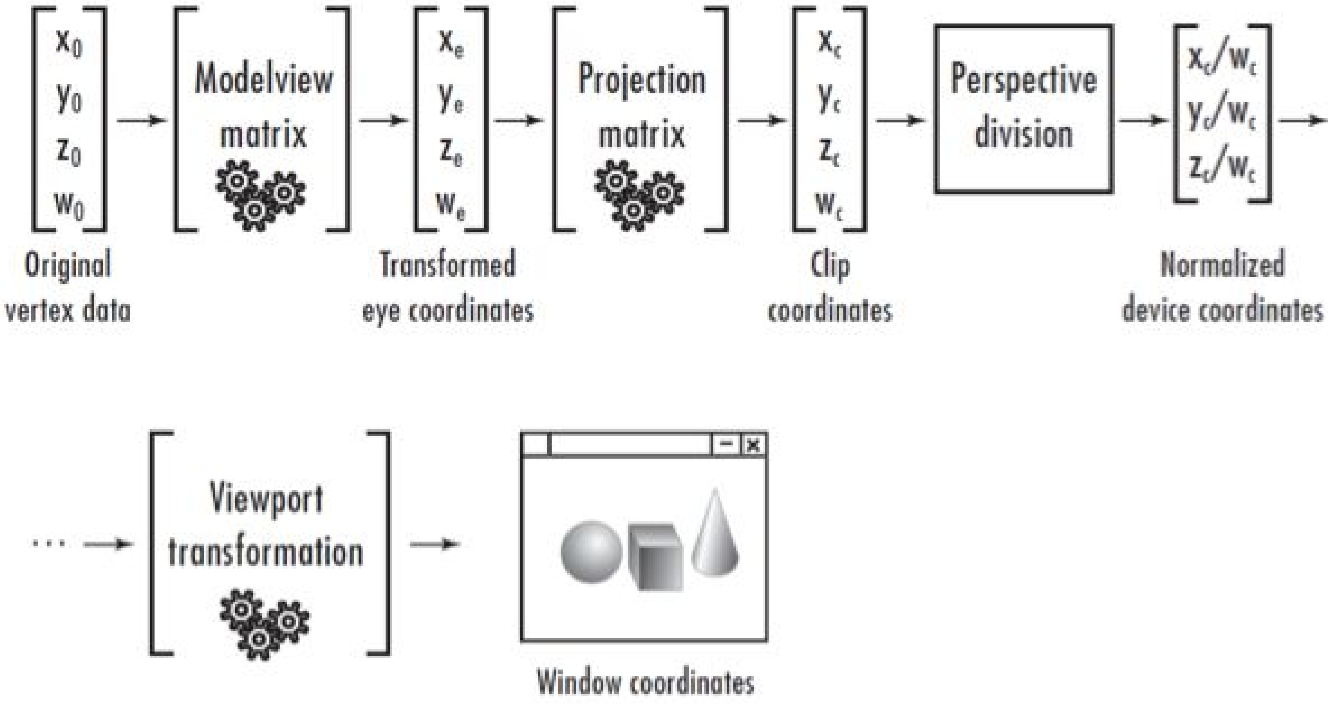
JOHANNES KEPLER UNIVERSITY LINZ
Institute of Computer Graphics

OpenGL provides a right-handed coordinate system

By default OpenGL's virtual camera is placed at the origin of this coordinate system looking in negative z-direction



Transformation Pipeline



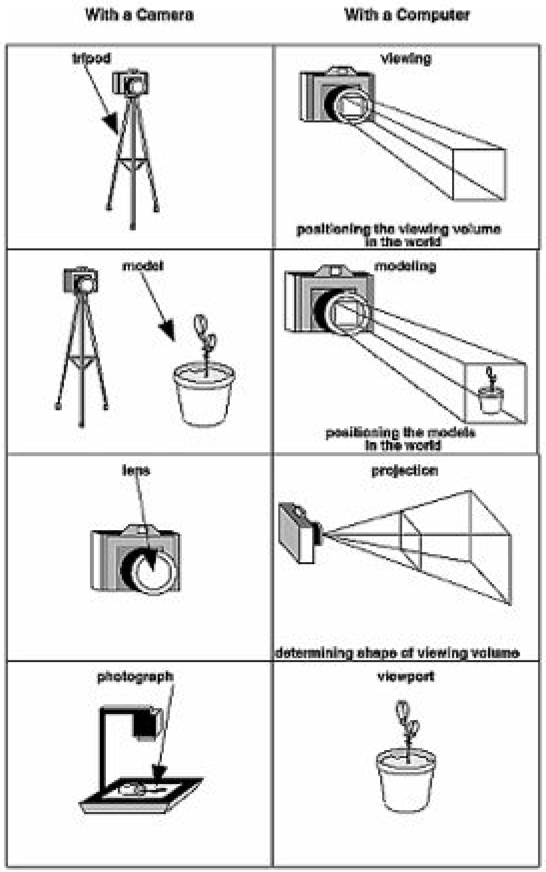
Transformation Pipeline

JOHANNES KEPLER UNIVERSITY LINZ
Institute of Computer Graphics

OpenGL follows a camera analogy

Think of

- the **view transformation** as placing a camera
- the **scene transformation** as placing an object
- the **projection transformation** as adjusting the camera's focus
- the **viewport transformation** as choosing the photograp



Matrices

All transformations are stored as 4x4 matrices

- Why use a 4x4 matrix for 3D?
- Remember homogenous coordinates?

Combine matrices and vectors by multiplying them

Identity matrix

- All 1 along diagonal, rest 0
- Neutral operation when multiplied with existing matrix or vector

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Transformation Pipeline

JOHANNES KEPLER UNIVERSITY LINZ
Institute of Computer Graphics

Scene and view transformations are considered the same in OpenGL

`modelViewMatrix = viewMatrix * sceneMatrix`

```
function setUpModelViewMatrix(viewMatrix, sceneMatrix) {

  var modelViewMatrix = matrixMultiply(viewMatrix, sceneMatrix );
  gl.uniformMatrix4fv(modelViewLocation, false, modelViewMatrix);
}
```

main.js

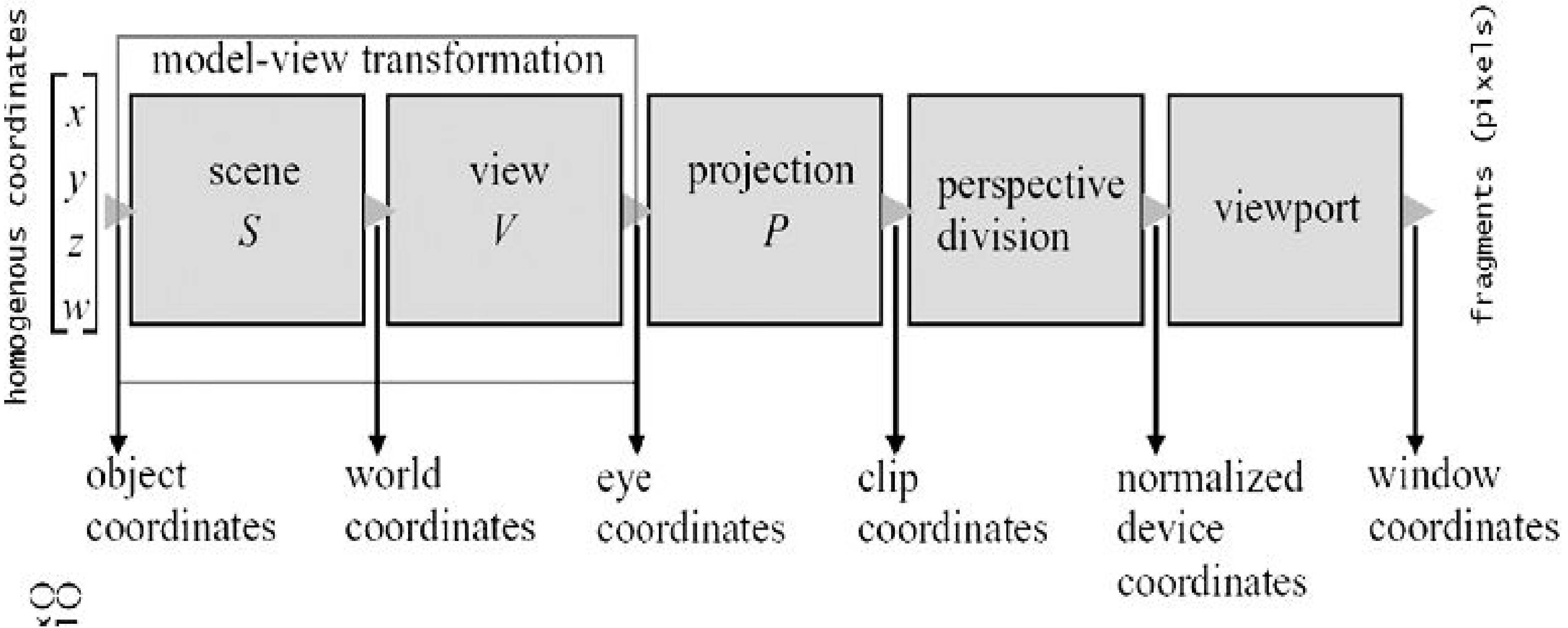
`projectionMatrix` multiplied in shader

All matrices in our framework are initialized with identity matrix

simple.vs.glsl

```
// the position of the point
attribute vec3 a_position;

//the color of the point
attribute vec3 a_color;

varying vec3 v_color;

uniform mat4 u_modelView;
uniform mat4 u_projection;

//like a C program main is the main function
void main() {

  gl_Position = u_projection * u_modelView
    * vec4(a_position, 1);

  //just copy the input color to the output varying color
  v_color = a_color;
}
```

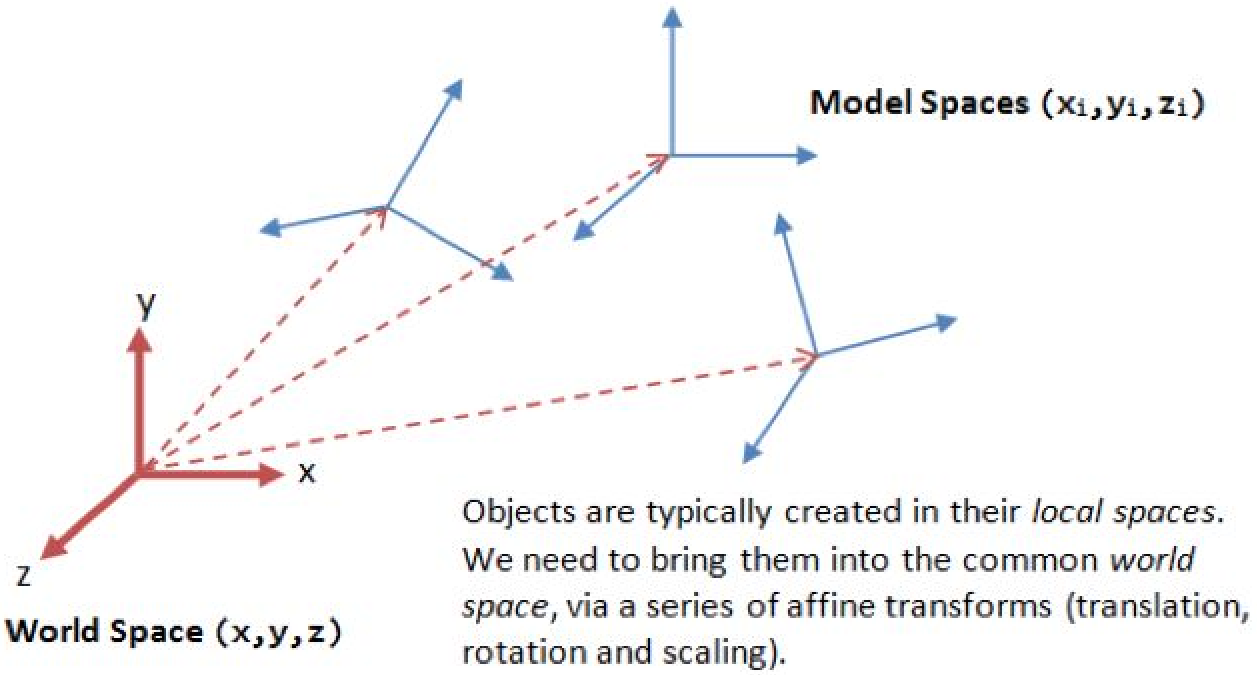
Transformation Pipeline



Model vs. World Space

JOHANNES KEPLER UNIVERSITY LINZ
Institute of Computer Graphics

Multiply model coordinates by scene matrix to get to world space

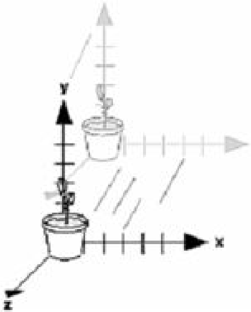


Transformations

Translation

Moves a point by a vector in x,y,z

See `makeTranslationMatrix()`

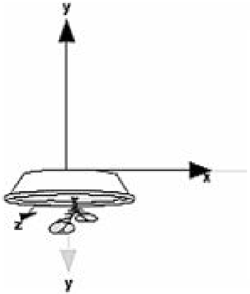


$$T = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling

Scales a point by a factor in x,y,z

See `makeScaleMatrix()`

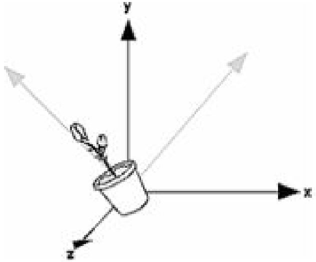


$$T = \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation

Rotates a point by degrees around x,y,z

See `makeX/Y/ZRotationMatrix()`



$$Rot_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos a & -\sin a & 0 \\ 0 & \sin a & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}; Rot_y = \begin{bmatrix} \cos a & 0 & \sin a & 0 \\ 0 & 1 & 0 & 0 \\ -\sin a & 0 & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}; Rot_z = \begin{bmatrix} \cos a & -\sin a & 0 & 0 \\ \sin a & \cos a & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

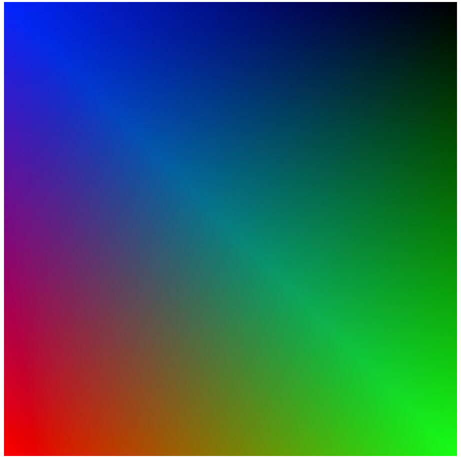
Task 1: Translation in Shader

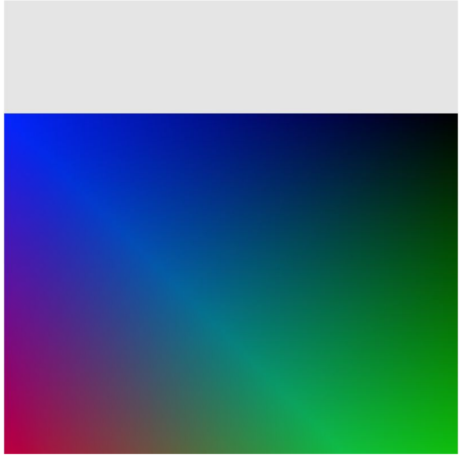
Goal: Move quad by -0.5 units in y-direction (down)

Step 1: Define 3D vector as local variable in vertex shader

```
vec3 v_translation = vec3(trans_x, trans_y, trans_z);
```

Step 2: Add translation vector to `a_position`





Task 1: Solution

```
// the position of the point
attribute vec3 a_position;

//the color of the point
attribute vec3 a_color;

varying vec3 v_color;

uniform mat4 u_modelView;
uniform mat4 u_projection;

//like a C program main is the main function
void main() {

  //TASK 1 and TASK 2-1
  //translation vector for moving vertices to a different position
  vec3 translation = vec3(0,-0.5,0);

  gl_Position = u_projection * u_modelView
    * vec4(a_position + translation, 1);

  //just copy the input color to the output varying color
  v_color = a_color;
}
```

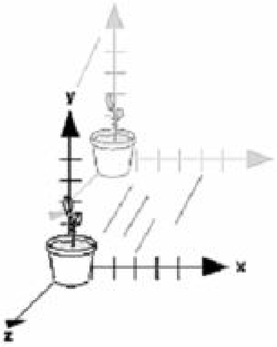
simple.vs.glsl

Task 2: Translation Using Matrix

Goal: Achieve same translation by manipulating modelView matrix
ModelView matrix already given as input to shader

Step 1: Remove translation in shader from last step
Step 2: Use `makeTranslationMatrix()` and set translation factors
Step 3: Multiply translation matrix with modelView matrix

Attention: Multiplication order!



$$T = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Task 2: Solution

```
function renderQuad(sceneMatrix, viewMatrix) {

  //TASK 2-2 and TASK 3 and TASK 4
  sceneMatrix = matrixMultiply(sceneMatrix, makeTranslationMatrix(0.0,-0.5,0));

  setUpModelViewMatrix(viewMatrix, sceneMatrix);
```

main.js

Transformation Order

Order matters!

Different result: `scale(translate(v))` vs. `translate(scale(v))`

Read from right to left

- Operations closest to the object definition are applied first

In OpenGL transformation commands are always issued in reverse order if multiple transforms are applied to a vertex

- Read code from bottom to top

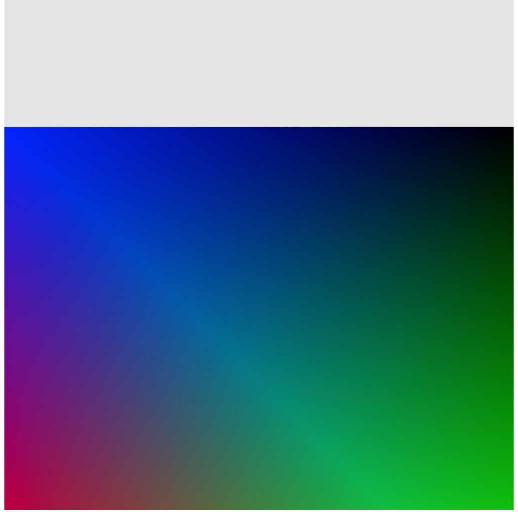
Task 3: Add Scaling to Matrix

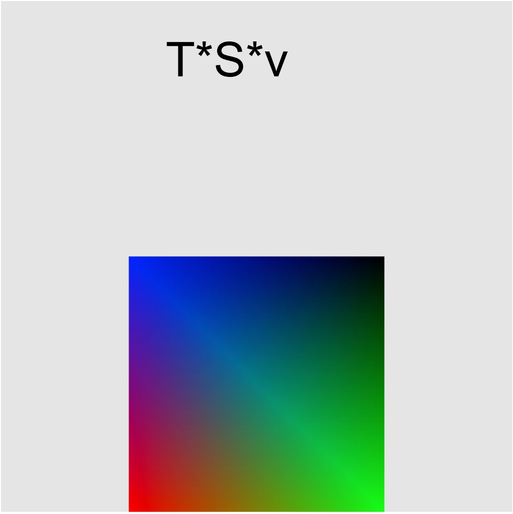
Goal: Shrink quad by 50% in x and y direction

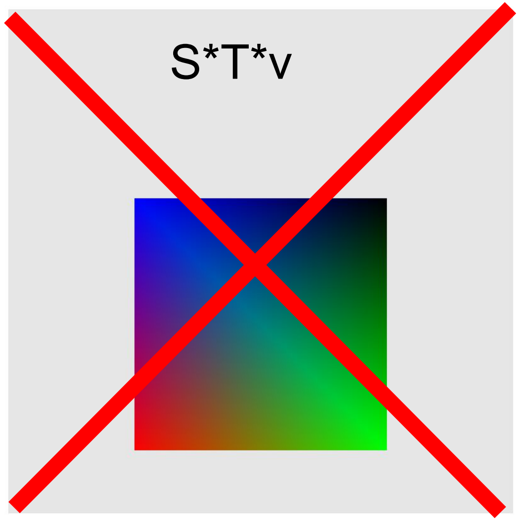
Step 1: Use `makeScaleMatrix()` function and set scale factors

Step 2: Multiply scale matrix with modelView matrix

Important: Do not scale translation (order!)







Task 3: Solution

```
function renderQuad(sceneMatrix, viewMatrix) {

  //TASK 2-2 and TASK 3 and TASK 4
  sceneMatrix = matrixMultiply( matrixMultiply(
                                sceneMatrix,
                                makeTranslationMatrix(0.0,-0.5,0) ),
                                makeScaleMatrix( .5, .5, 1) );

  setUpModelViewMatrix(viewMatrix, sceneMatrix);
```

OR

```
function renderQuad(sceneMatrix, viewMatrix) {

  //TASK 2-2 and TASK 3 and TASK 4
  sceneMatrix = matrixMultiply(sceneMatrix, makeTranslationMatrix(0.0,-0.5,0));
  sceneMatrix = matrixMultiply(sceneMatrix, makeScaleMatrix( .5, .5, 1));

  setUpModelViewMatrix(viewMatrix, sceneMatrix);
```

main.js

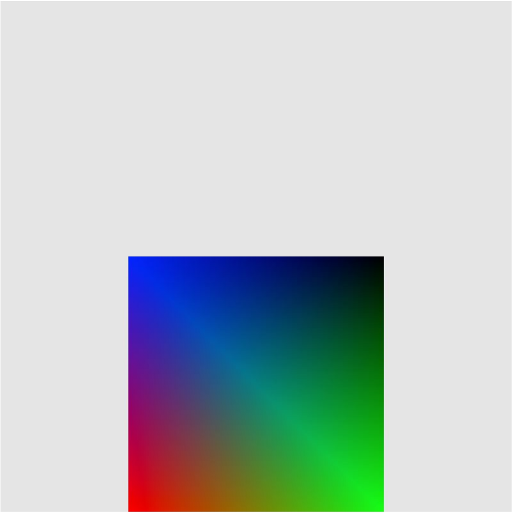
Task 4: Add Rotation

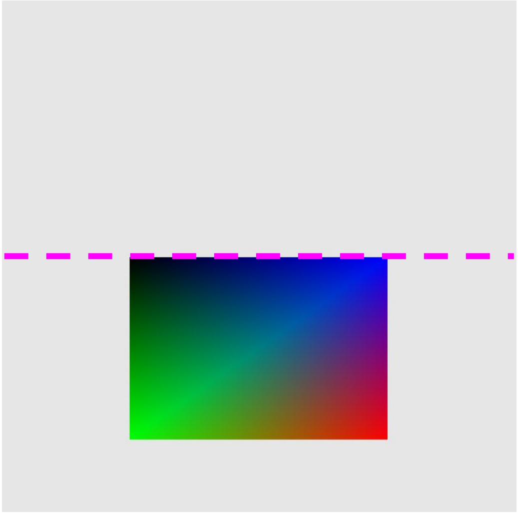
Goal: Rotate quad around x-axis by 45 degrees

Step 1: Use `makeXRotationMatrix()` and set rotation factors

Step 2: Multiply rotation matrix with modelView matrix

Important: Think about order!





Rotations around x-axis

Task 4: Solution

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

```
function renderQuad(sceneMatrix, viewMatrix) {

  //TASK 2-2 and TASK 3 and TASK 4
  sceneMatrix = matrixMultiply( matrixMultiply( matrixMultiply(
                                sceneMatrix,
                                makeXRotationMatrix(convertDegreeToRadians(45)) ),
                                makeTranslationMatrix(0.0,-0.5,0) ),
                                makeScaleMatrix( .5, .5, 1) );

  setUpModelViewMatrix(viewMatrix, sceneMatrix);
```

OR

```
function renderQuad(sceneMatrix, viewMatrix) {

  //TASK 2-2 and TASK 3 and TASK 4
  sceneMatrix = matrixMultiply(sceneMatrix, makeXRotationMatrix(convertDegreeToRadians(45)));
  sceneMatrix = matrixMultiply(sceneMatrix, makeTranslationMatrix(0.0,-0.5,0));
  sceneMatrix = matrixMultiply(sceneMatrix, makeScaleMatrix( .5, .5, 1));

  setUpModelViewMatrix(viewMatrix, sceneMatrix);
```

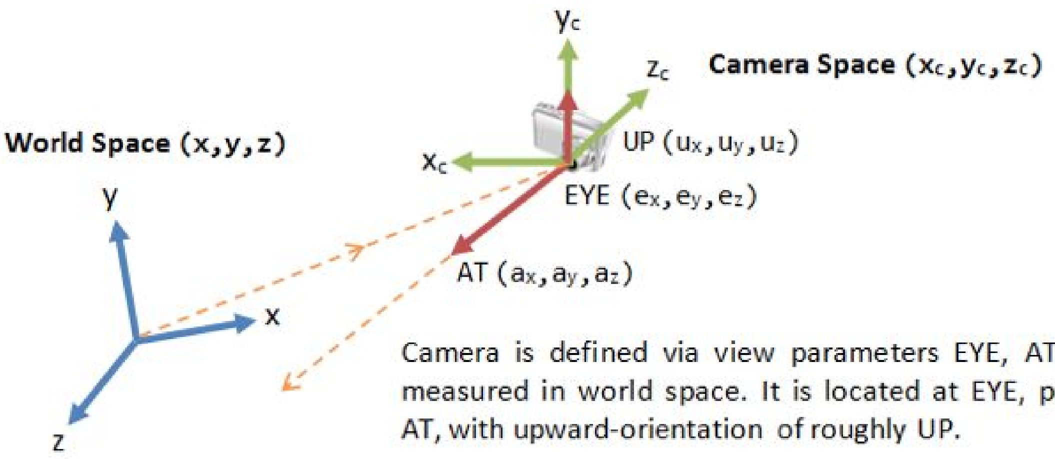
main.js

View Transformations (Camera)

Recalling the camera analogy, viewing transformations position and point the camera towards our scene

Scene and view transformation considered the same in OpenGL

- Think of moving the camera or the whole scene



Camera is defined via view parameters EYE, AT and UP, measured in world space. It is located at EYE, pointing at AT, with upward-orientation of roughly UP.

In the Camera space, camera is located at origin, pointing at $-z_c$, with upward-orientation of y_c.
z_c is opposite of AT, y_c is roughly UP.

View Transformations (Camera)

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

There are different ways to change viewing direction and vantage point

Option 1:

Use translate and rotate operations to change viewpoint (i.e., moving all objects)

Option 2:

Create and use lookAt matrix

It specifies the viewpoint, viewing direction and up-vector (i.e., camera’s rotation)

Note that you can have only one view transformation!

lookAt-Matrix Example

Bob is hanging upside down from a branch, looking at Alice, lying on the grass with a book.

```
lookAt(Bob_x, Bob_y, Bob_z, Alice_x, Alice_y, Alice_z,
UpVector_x, UpVector_y, UpVector_z );
```

Bob‘s branch is at (20,80,15) (it‘s a tall tree)

Alice is at (15,0,12) (near the foot of the tree)

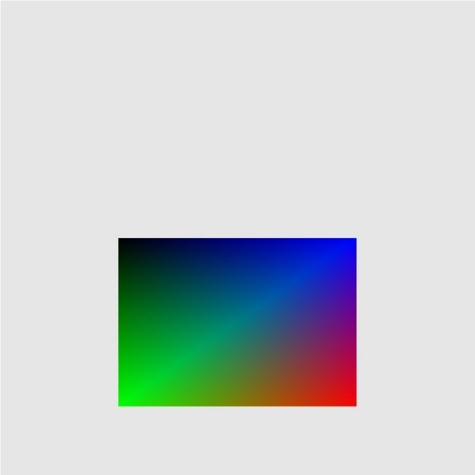
Upside-down mean your up-vector is (0,-1,0)

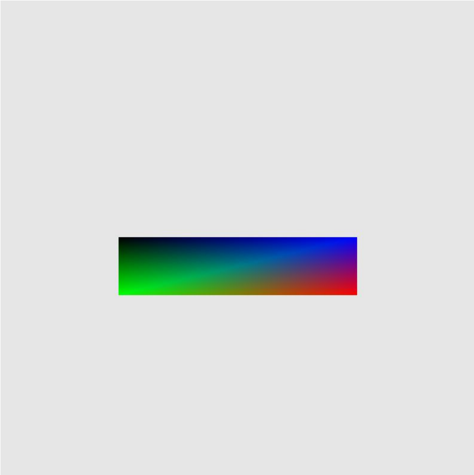
```
lookAt( 20, 80, 15, 15, 0, 12, 0, -1, 0 );
```

Task 5: Setup lookAt Camera

Goal: Let camera look at origin from position (0,3,5)

Step 1: Call `lookAt()` function in `calculateViewMatrix()`





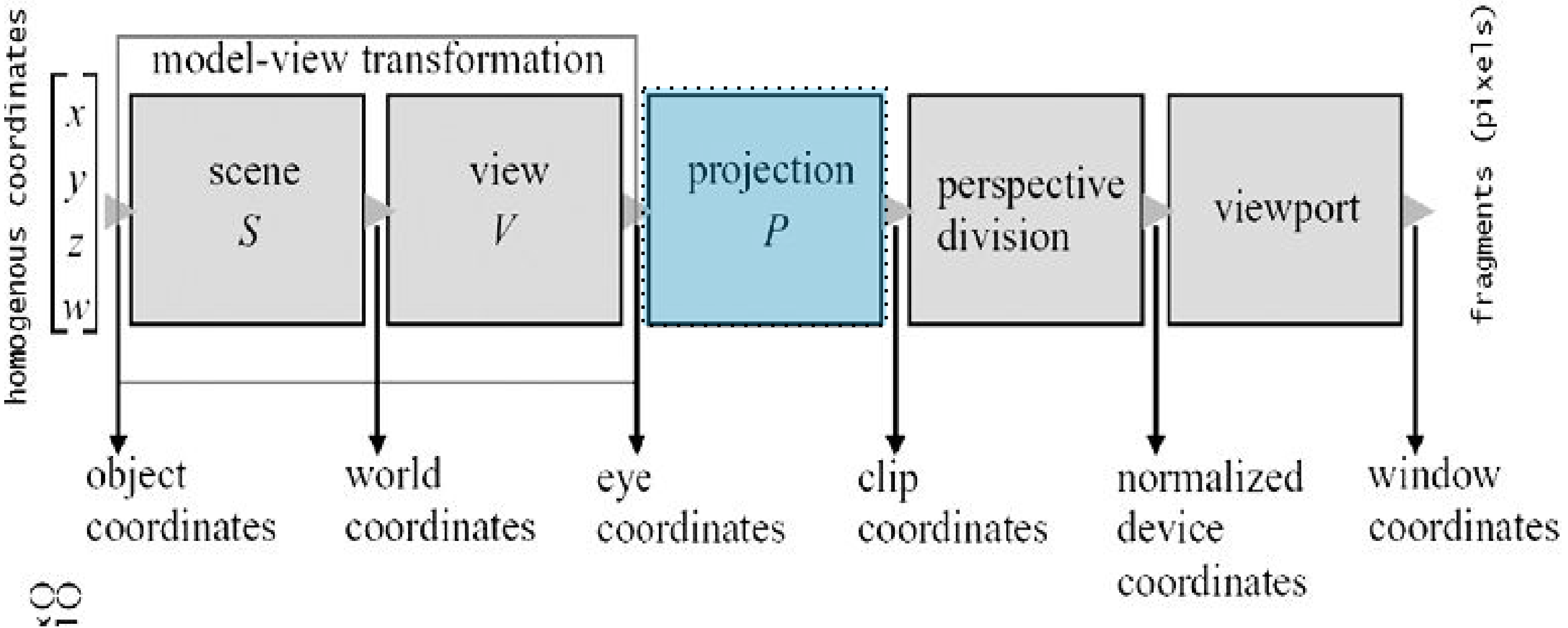
Task 5: Solution

```
function calculateViewMatrix(viewMatrix) {
  //compute the camera's matrix
  // TASK 5
  viewMatrix = lookAt(0,3,5,0,0,0,0,1,0);
  return viewMatrix;
}
```

viewer, origin, up-vector

main.js

Projective Transformations

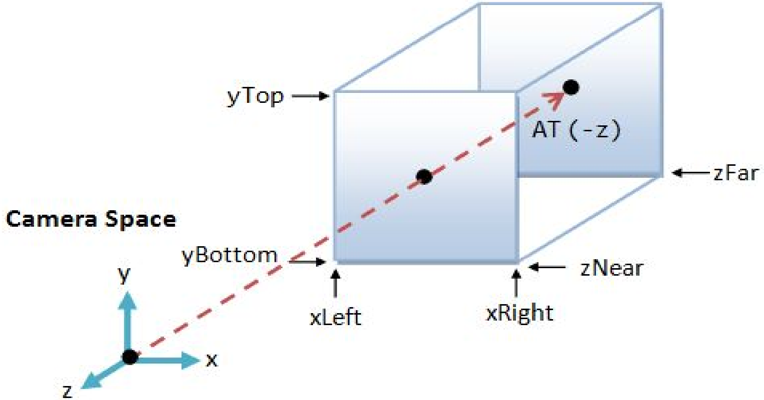


Projective Transformations

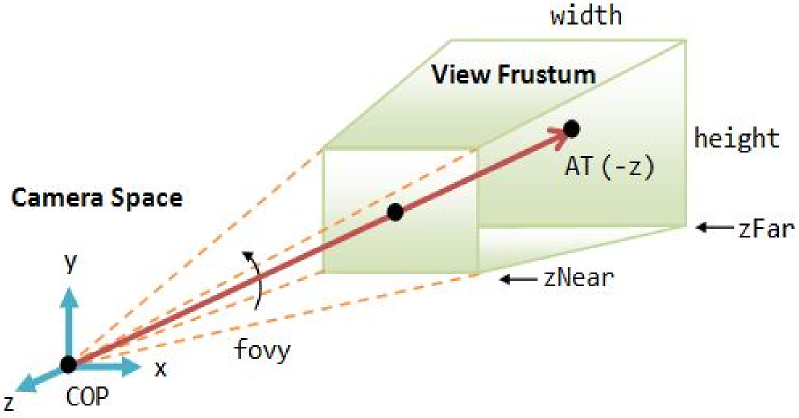
Projection transf. are like choosing our camera’s lens or field of view

Used to describe a viewing volume and how objects are projected

Projections may be either perspective or orthographic



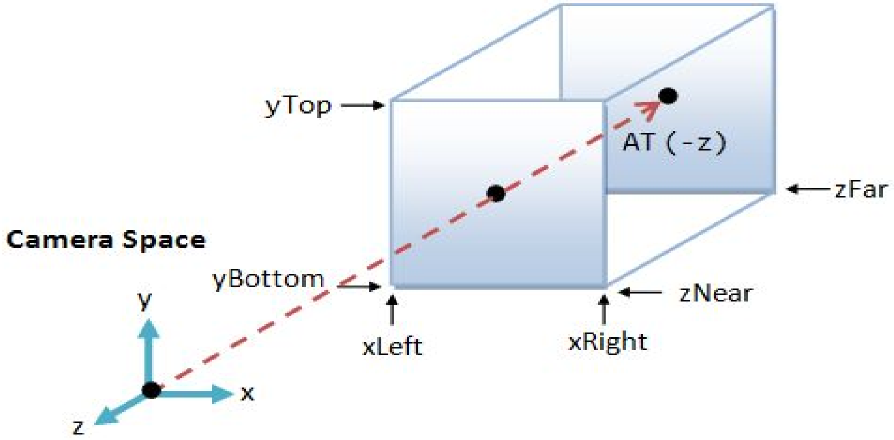
Orthographic Projection: Camera positioned infinitely far away at $z = \infty$



Orthographic Projection

Orthographic projections require a box shaped viewing volume

`makeOrthographicProjectionMatrix(left,right,bottom,top,near,far)`



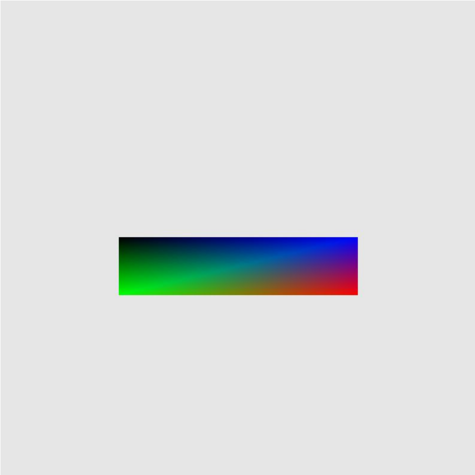
Orthographic Projection: Camera positioned infinitely far away at $z = \infty$

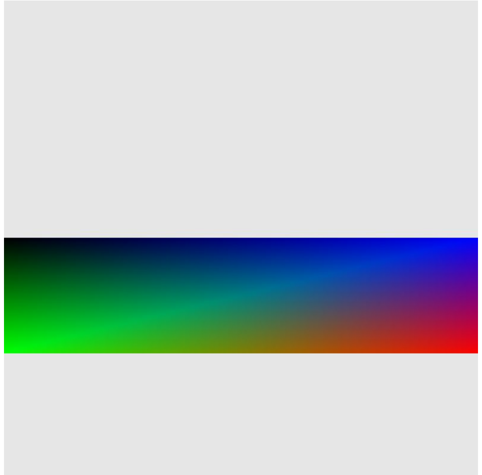
TASK 6: Orthographic Projection

Goal: Set up orthographic projection

Step 1: Call `makeOrthographicProjectionMatrix(left,right,bottom,top,near,far)`

With settings: `left=-0.5, right=0.5, bottom=-0.5, top=0.5, near=0, far=10`





Task 6: Solution

```
var projectionMatrix = defaultProjectionMatrix;
// TASK 6
projectionMatrix = makeOrthographicProjectionMatrix(-.5,.5,-.5,.5,0,10);
// TASK 7

gl.uniformMatrix4fv(projectionLocation, false, projectionMatrix);
```

main.js

Perspective Projection

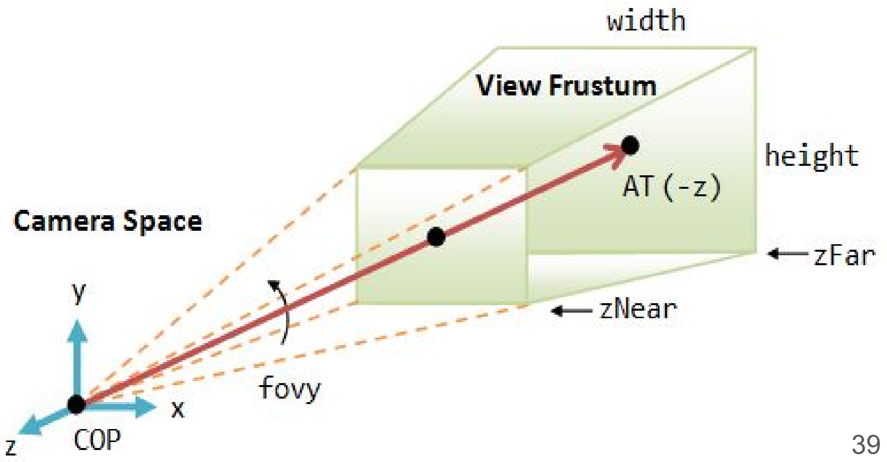
Perspective projections require a frustum shaped viewing volume

- Truncated section of a pyramid

Two options to define a frustum:

- Specify left, right, bottom, top, distance of near and far clipping plane
  - OR
- Specify field of view (angle), aspect ratio (width/height), distance of near and far clipping plane

We will use the second option.



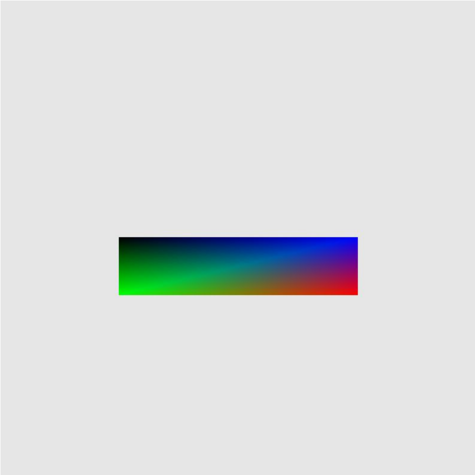
TASK 7: Perspective Projection

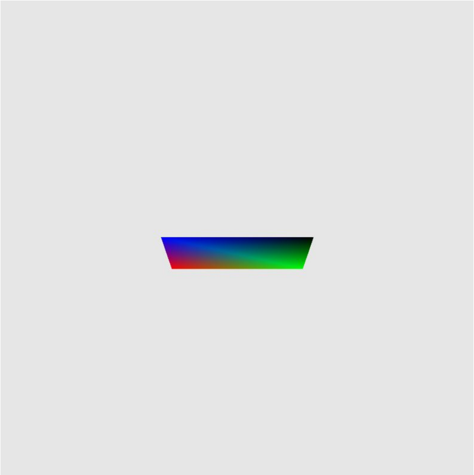
Goal: Set up perspective projection

Step 1: `makePerspectiveProjectionMatrix(fieldOfViewInRadians,aspect,near,far)`

With settings: `fieldOfViewInRadians=30 degree, aspectRatio=canvasWidth/canvasHeight, near=1, far=10`

You'll notice perspective foreshortening



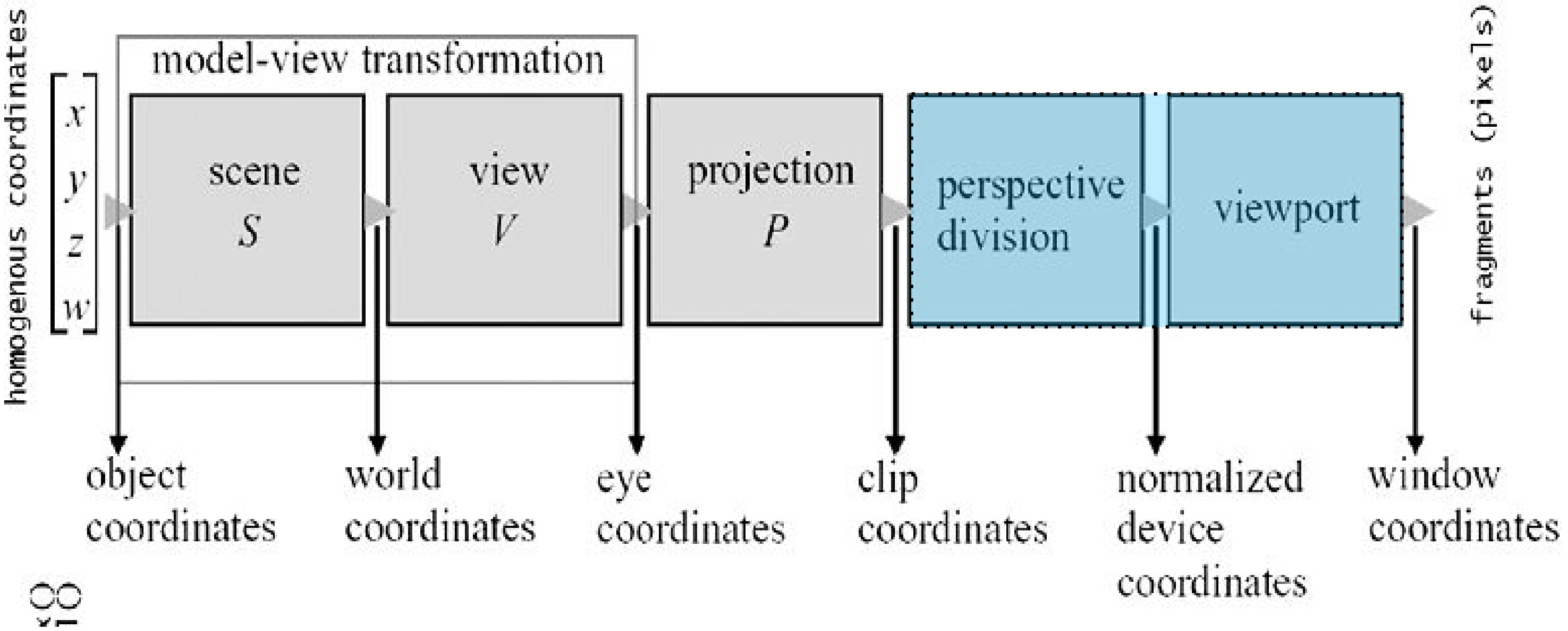


Task 7: Solution

```
 var projectionMatrix = defaultProjectionMatrix;
// TASK 6

// TASK 7
projectionMatrix = makePerspectiveProjectionMatrix(fieldOfViewInRadians,
    aspectRatio, 1, 10 );

gl.uniformMatrix4fv(projectionLocation, false, projectionMatrix);
```

main.js

Perspective Division & Viewport



Perspective Division & Viewport

This step is independent from the user, it cannot be affected

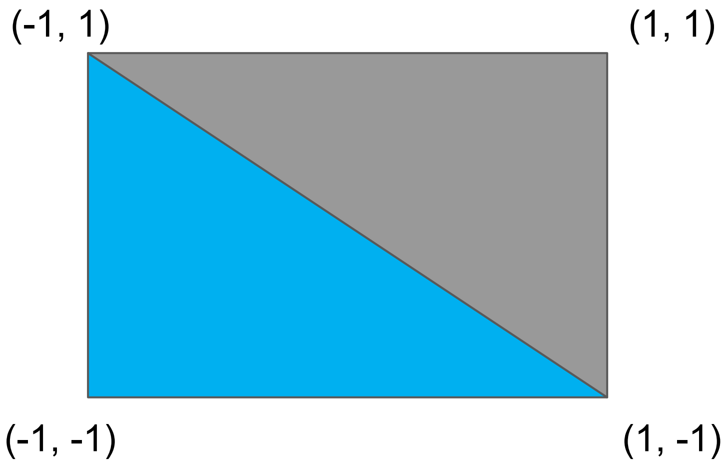
Vertex coordinates are being divided by the w-coordinate and we obtain normalized device coordinates (NDC) ranging from -1 to 1 in x, y

The z-coordinate (depth) is treated as always ranging from 0.0 to 1.0

There’s more on depth handling in our next exercise!

Reusing Vertices via Index Buffer

Quad from Lab 1 consists of 2 triangles
Drawback: Some vertices need to be send to GPU multiple times
Instead of defining vertices multiple times indexing can be used




```
const arr = new Float32Array([
    -1.0, -1.0,
     1.0, -1.0,
    -1.0,  1.0,
    -1.0,  1.0,
     1.0, -1.0,
     1.0,  1.0
]);
```

Task 8: Add Cube

Cube geometry defined at top

`cubeVertices, cubeColors, cubeIndices`

Step 1: Initialize buffers by calling `initCubeBuffer()`

Step 2: call `renderRobot()`

Step 3: Render cube by calling `renderCube()` in `renderRobot()`

Task 8-1 and 8-2: Solution

```
  // TASK 8-1
  //set buffers for cube
  initCubeBuffer();
}
```

```
  renderQuad(sceneMatrix, viewMatrix);

  // TASK 8-2
  renderRobot(sceneMatrix, viewMatrix);

  //request another render call as soon as possible
  requestAnimationFrame(render);
```

main.js

Task 8-3: Solution

```
function renderRobot(sceneMatrix, viewMatrix) {

  gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexBuffer);
  gl.vertexAttribPointer(positionLocation, 3, gl.FLOAT, false,0,0) ;
  gl.enableVertexAttribArray(positionLocation);

  gl.bindBuffer(gl.ARRAY_BUFFER, cubeColorBuffer);
  gl.vertexAttribPointer(colorLocation, 3, gl.FLOAT, false,0,0) ;
  gl.enableVertexAttribArray(colorLocation);

  // TASK 10-2

  // store current sceneMatrix in originSceneMatrix, so it can be restored
  var originSceneMatrix = sceneMatrix;

  // TASK 9 and 10

  setUpModelViewMatrix(viewMatrix, sceneMatrix);
  // TASK 8-3
  renderCube();

  // TASK 10-1

}
```

main.js

Task 9: Create Animation

Goal: Rotate cube

Principle: Apply small transformations in every render call

Independent of the frame rate:

```
function render(timeInMilliseconds) {
```

```
animatedAngle = timeInMilliseconds/10;
```

Step 1: add rotation around y-axis of cube

by using variable: `animatedAngle`

Task 9: Solution

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

```
function renderRobot(sceneMatrix, viewMatrix) {

  gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexBuffer);
  gl.vertexAttribPointer(positionLocation, 3, gl.FLOAT, false,0,0) ;
  gl.enableVertexAttribArray(positionLocation);

  gl.bindBuffer(gl.ARRAY_BUFFER, cubeColorBuffer);
  gl.vertexAttribPointer(colorLocation, 3, gl.FLOAT, false,0,0) ;
  gl.enableVertexAttribArray(colorLocation);

  // TASK 10-2

  // store current sceneMatrix in originSceneMatrix, so it can be restored
  var originSceneMatrix = sceneMatrix;

  // TASK 9 and 10
  sceneMatrix = matrixMultiply(sceneMatrix, makeYRotationMatrix(convertDegreeToRadians(animatedAngle)));
  setUpModelViewMatrix(viewMatrix, sceneMatrix);
  renderCube();

  // TASK 10-1

}
```

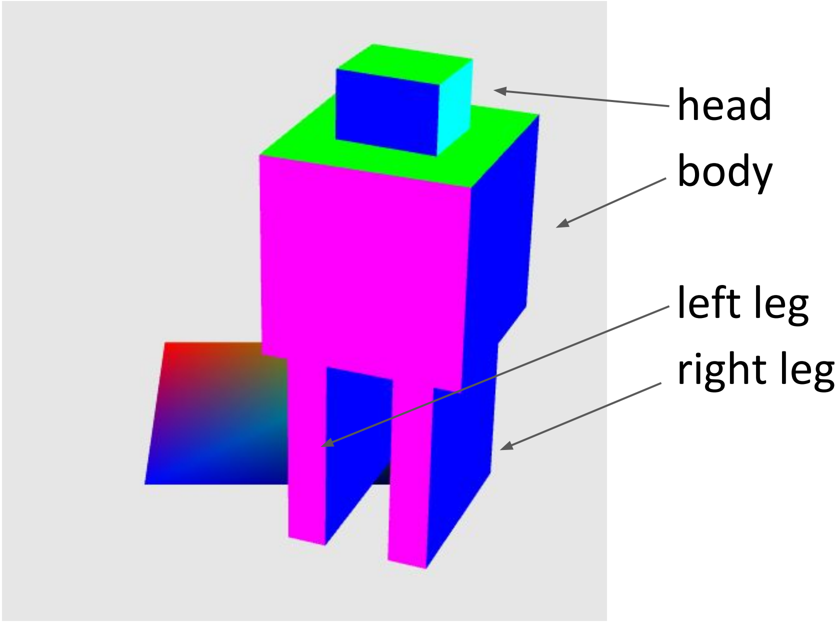
main.js

Let’s Create a Robot

Robot should stand on ground plane (our quad)

Build robot from 4 cubes

Transform (translate,scale,rotate) cubes



Task 10: Complex Transformations

Goal: Create robot with rotating head that walks circles on ground

Step 0: Make ground plane (rotate quad by 90°)

Step 1: Create robot by adding cube multiple times

- Body, head, left leg, right leg
- rotating cube is the robot's head

Step 2: Let robot walk circles (without moving the legs)

Task 10-1: Solution

```
// TASK 9 and 10
sceneMatrix = matrixMultiply(sceneMatrix, makeYRotationMatrix(convertDegreeToRadians(animatedAngle)));
sceneMatrix = matrixMultiply(sceneMatrix, makeTranslationMatrix(0.0,0.4,0));
sceneMatrix = matrixMultiply(sceneMatrix, makeScaleMatrix(0.4,0.33,0.5));
setUpModelViewMatrix(viewMatrix, sceneMatrix);
renderCube();

// TASK 10-1
//body
sceneMatrix = originSceneMatrix;
setUpModelViewMatrix(viewMatrix, sceneMatrix);
renderCube();

//left leg
sceneMatrix = originSceneMatrix;
sceneMatrix = matrixMultiply(sceneMatrix, makeTranslationMatrix(0.16,-0.6,0));
sceneMatrix = matrixMultiply(sceneMatrix, makeScaleMatrix(0.2,1,1));
setUpModelViewMatrix(viewMatrix, sceneMatrix);
renderCube();

//right leg
sceneMatrix = originSceneMatrix;
sceneMatrix = matrixMultiply(sceneMatrix, makeTranslationMatrix(-0.16,-0.6,0));
sceneMatrix = matrixMultiply(sceneMatrix, makeScaleMatrix(0.2,1,1));
setUpModelViewMatrix(viewMatrix, sceneMatrix);
renderCube();
```

main.js

Task 10-2: Solution

```
// TASK 10-2
// transformations on whole body
sceneMatrix = matrixMultiply(sceneMatrix, makeYRotationMatrix(convertDegreeToRadians(animatedAngle/2)));
sceneMatrix = matrixMultiply(sceneMatrix, makeTranslationMatrix(0.3,0.9,0));

// store current sceneMatrix in originSceneMatrix, so it can be restored
var originSceneMatrix = sceneMatrix;


// TASK 9 and 10
sceneMatrix = matrixMultiply(sceneMatrix, makeYRotationMatrix(convertDegreeToRadians(animatedAngle)));
sceneMatrix = matrixMultiply(sceneMatrix, makeTranslationMatrix(0.0,0.4,0));
sceneMatrix = matrixMultiply(sceneMatrix, makeScaleMatrix(0.4,0.33,0.5));
setUpModelViewMatrix(viewMatrix, sceneMatrix);
```

main.js

Recap

Transformation pipeline

Model-view transformations

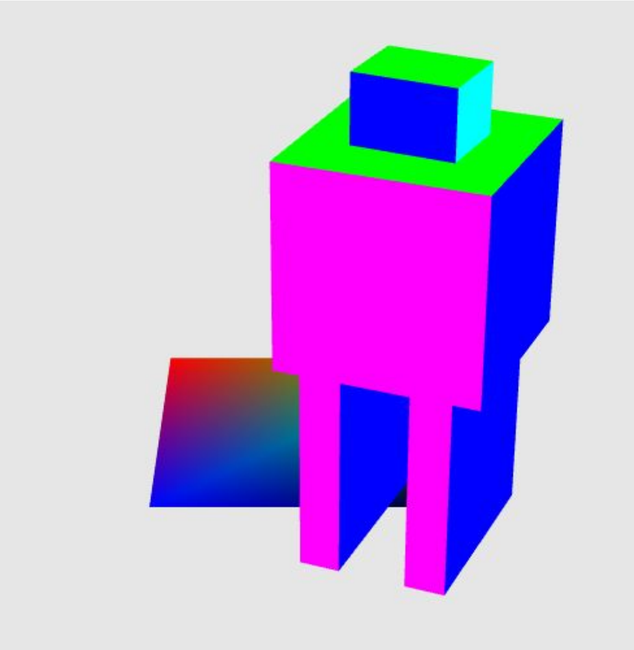
- Translate, scale, rotate, animations

Creating geometry using the index buffer

Projective transformations

- Orthographic and perspective projection

Camera transformations



Next Time

Rendering multiple objects

Blending and depth handling

Scene graph nodes and traversal

glMatrix JavaScript library

- Replaces matrix specific functions at the end of main.js from Lab 2 (e.g., multiply, lookAt, inverse …)

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computer Graphics

Practice at Home!

Institute of
Computer Graphics