JOHANNES KEPLER
UNIVERSITY LINZ

INSTITUTE OF
COMPUTER GRAPHICS

Computer Graphics

Lab 3: Scene Graphs

Lab Schedule

Lab 1	Introduction to WebGL	Week from March 8
Lab 2	Transformations and Projections	Week from March 15
Lab 3	**Scene Graphs**	**Week from March 22**
Lab 4	Illumination and Shading	Week from April 12
Lab 5	Texturing	Week from April 19
Lab 6	Advanced Texture Mapping	Week from April 26
Lab 7a	Project Q&A	1.6. 15:30-17:00, 11.6. 08:30-10:00
Lab 7b	Introduction to CUDA	11.6. 10:00-11:30 & 12:15-13:45

CG Lab Project: Create a Movie

Group project in teams of 2 students

Mandatory Submissions via Github:

- **26.03.2021 23:59:** Movie concept submission (incl. team announcement)
- **23.04.2021 23:59:** Intermediate submission
- **22.06.2021 23:59:** Hand-in final package

Individual interviews (alone): **24.-30.06.2021**

Fork me on GitHub

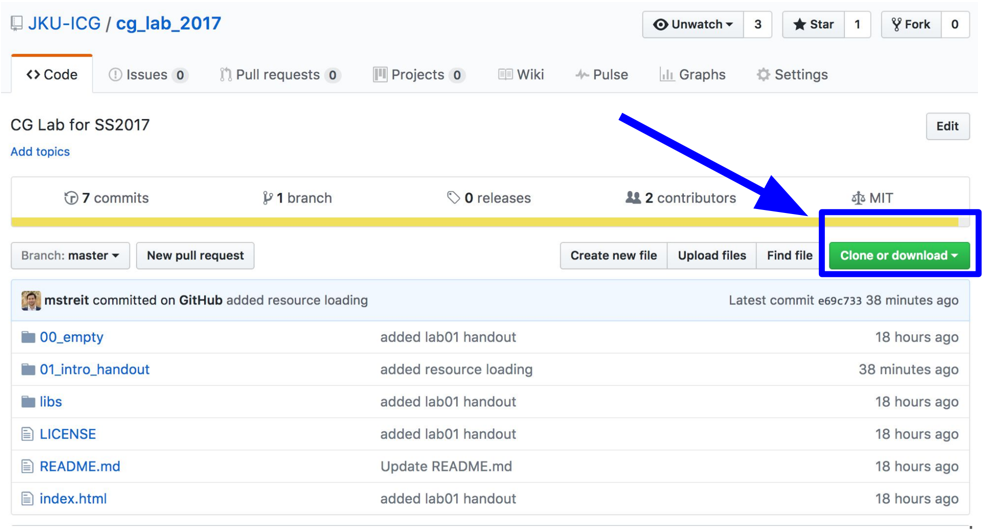
Dev Environment: Lab Package

Hosted on GitHub: https://github.com/jku-icg/cg_lab_2021

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 Visual Studio Code
4. Open `cg_lab_2021` folder (*File → Open*)
5. Click on **Go Live** button in lower right corner



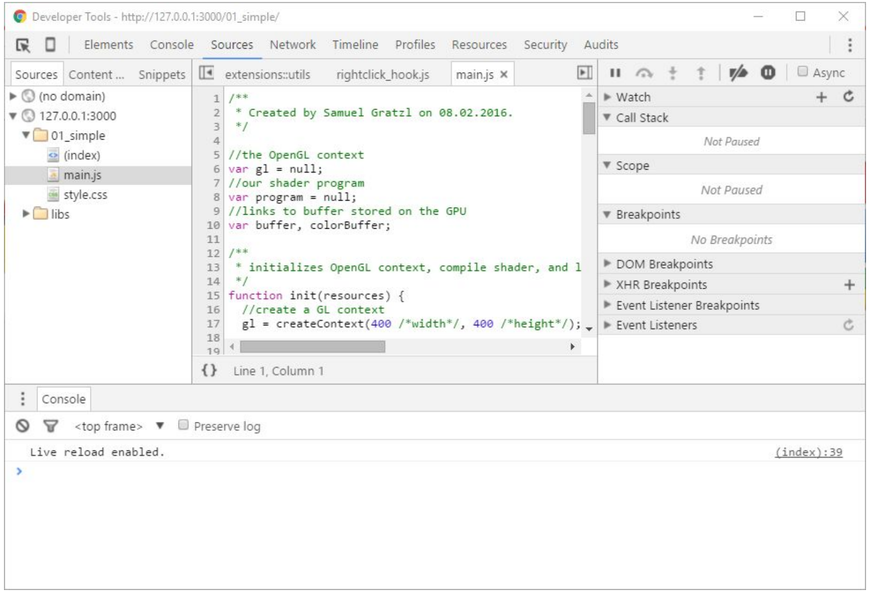
CG Lab for SS2017

mstreit committed on GitHub added resource loading — Latest commit e69c733 38 minutes ago

00_empty	added lab01 handout	18 hours ago
01_intro_handout	added resource loading	38 minutes ago
libs	added lab01 handout	18 hours ago
LICENSE	added lab01 handout	18 hours ago
README.md	Update README.md	18 hours ago
index.html	added lab01 handout	18 hours ago

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, …



```
/**
 * Created by Samuel Gratzl on 08.02.2016.
 */

//the OpenGL context
var gl = null;
//our shader program
var program = null;
//links to buffer stored on the GPU
var buffer, colorBuffer;

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

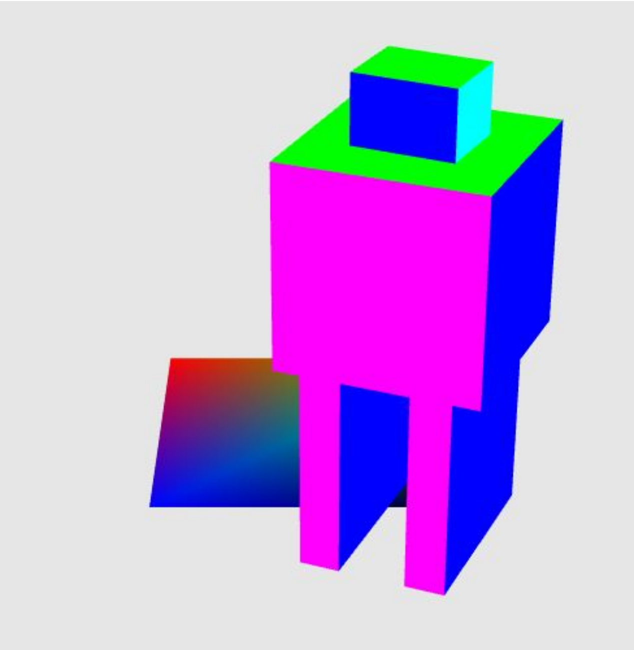
```
Live reload enabled.
```

Recap

- Transformation pipeline
- Model-view transformations
  - Translate, scale, rotate
- Creating geometry using the index buffer
- Projective transformations
  - Orthographic and perspective projection
- Camera transformations
- Animations
- Solution on GitHub:

02_transformations

02_transformations_handout



glMatrix Helper Library

Library for vector and matrix manipulation: http://glmatrix.net/

Library supports:

Identity matrix, multiplication, inverse, clone, translation, rotation, scale, lookAt, orthographic & perspective projection, etc.

Already part of framework: `libs/gl-matrix.js`

Documentation: http://glmatrix.net/docs/

Use it from now on!

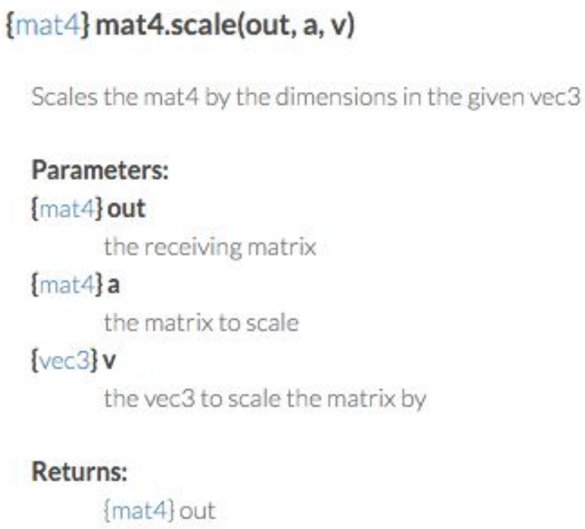
glMatrix Usage

Classes vec2, vec3, vec4, mat2, mat3, mat4, etc.

```
var identity = mat4.create();
var out = mat4.scale(mat4.create(),identity,[x, y, z]);
```

For convenience in our framework:

```
var out = glm.scale(x,y,z)
```



{mat4} **mat4.scale(out, a, v)**

Scales the mat4 by the dimensions in the given vec3

Parameters:

{mat4} **out**
the receiving matrix

{mat4} **a**
the matrix to scale

{vec3} **v**
the vec3 to scale the matrix by

Returns:

{mat4} out

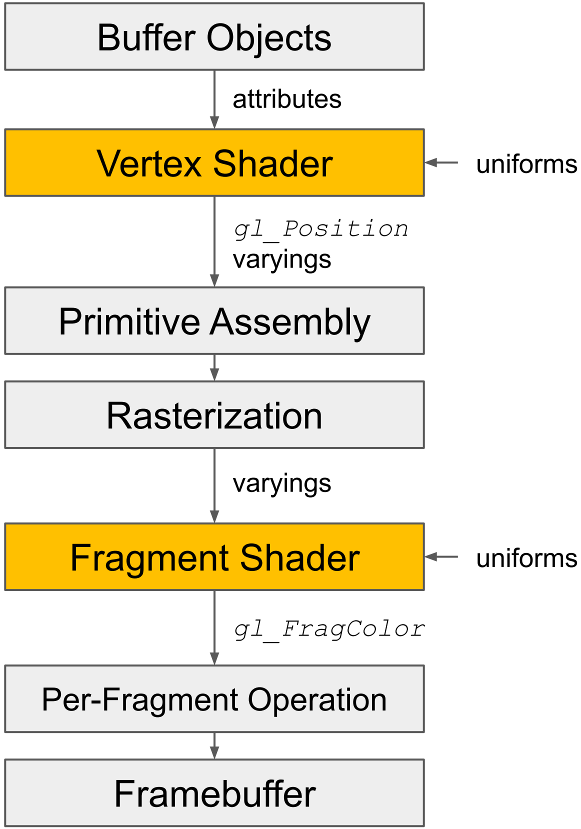
Recap: Structure of a WebGL Program

JOHANNES KEPLER UNIVERSITY LINZ
INSTITUTE OF COMPUTER GRAPHICS

- At initialization time `init()`
  - Create all shaders and programs
  - Create buffers and upload vertex data
- At render time `render()`
  - Set global states (enable depth testing, etc.)
  - For each object you want to draw
    - Call `gl.useProgram` for the program needed to draw
    - Setup attributes for the object you want to draw
    - For each attribute call `gl.bindBuffer`, `gl.vertexAttribPointer`, `gl.enableVertexAttribArray`
    - Setup uniforms for the object you want to draw by calling `gl.uniformXXX`
    - Call `gl.drawArrays` or `gl.drawElements`

http://webglfundamentals.org/webgl/lessons/webgl-drawing-multiple-things.html

Recap: Programmable Pipeline



Summary

vertex: point in 2D/3D space

fragment: pixel + additional properties

shader: tiny program on the GPU

shader program: vertex + fragment shader

buffer: array on GPU

attribute: accessing the current buffer element in shader

uniform: parameter from program to shader

varying: parameter between shader

gl_Position, gl_FragColor: magic variables

rasterization: 3 vertices → N fragments

JOHANNES KEPLER
UNIVERSITY LINZ

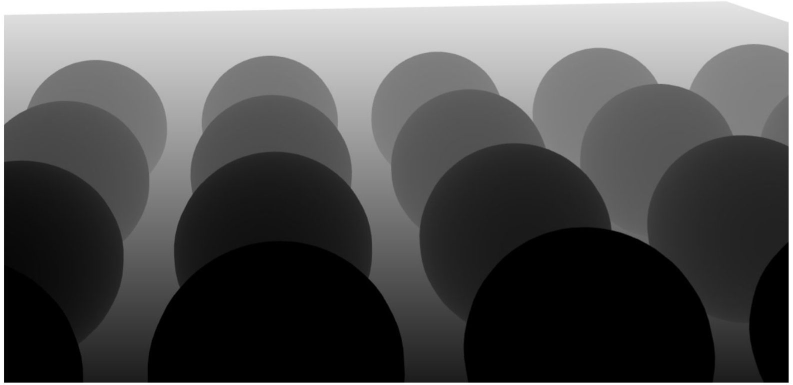
INSTITUTE OF
COMPUTER GRAPHICS

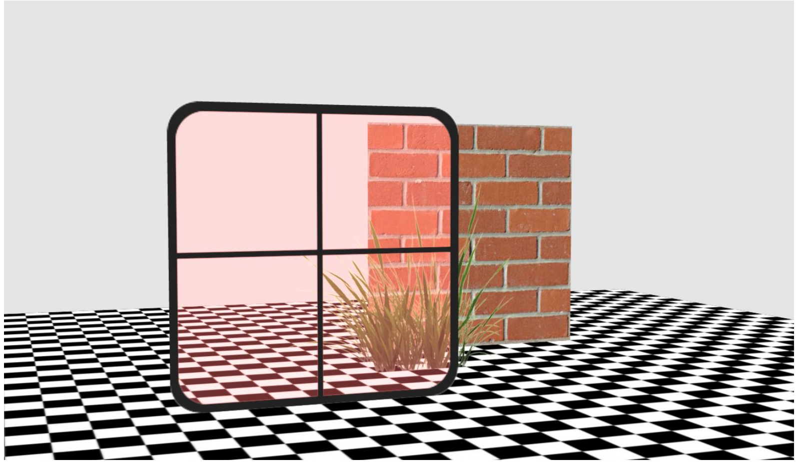
Computer Graphics

Lab 3: Per-Fragment Operations

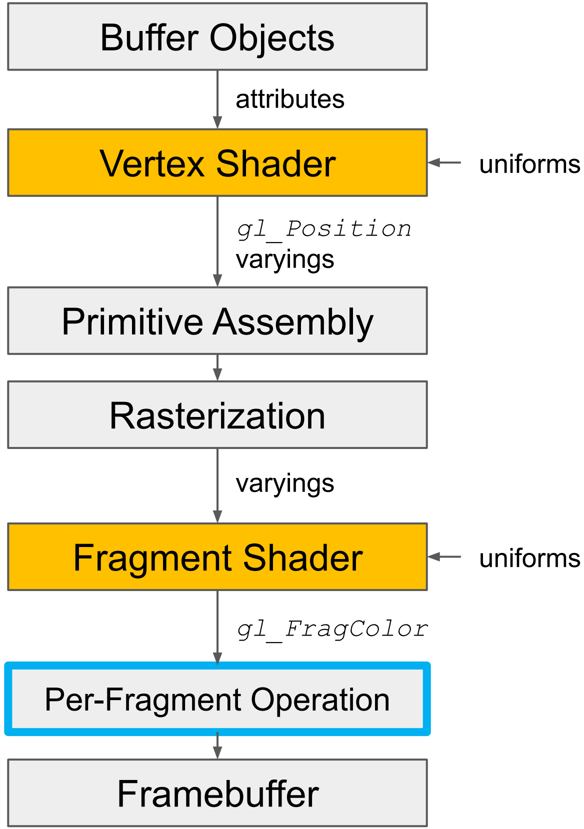
Agenda for This Week

- Per-Fragment Operations
  - Depth Handling
  - Blending
  - Tutorial (coding)
- Scene Graphs
  - Abstraction into Nodes
  - Scene graph traversal
  - Base node class
  - Render nodes
  - Transformation nodes
  - Implement robot using a scene graph
  - Tutorial (coding)





Programmable Pipeline



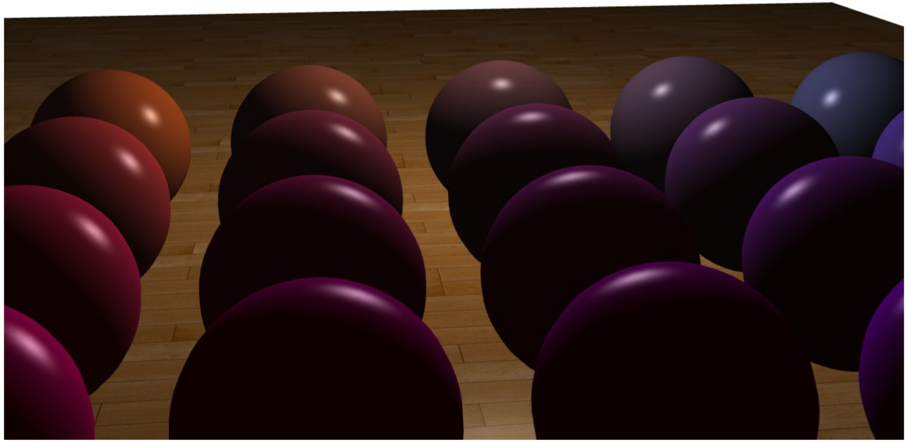
Per-Fragment Operation

- additional tests/operations per fragment
- decides if (and how) fragments are written into the framebuffer
- examples:
    - depth test
      the closer fragment should be drawn
    - blending
      in case of semi transparent fragments
    - ...

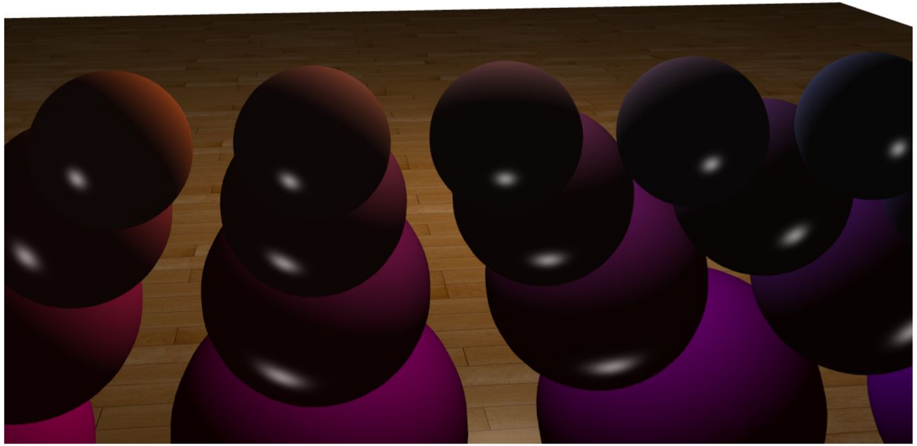
Depth Handling

Allows OpenGL to decide which object is in front

Depth testing has to be enabled to tell OpenGL to perform a depth test and use the depth buffer.



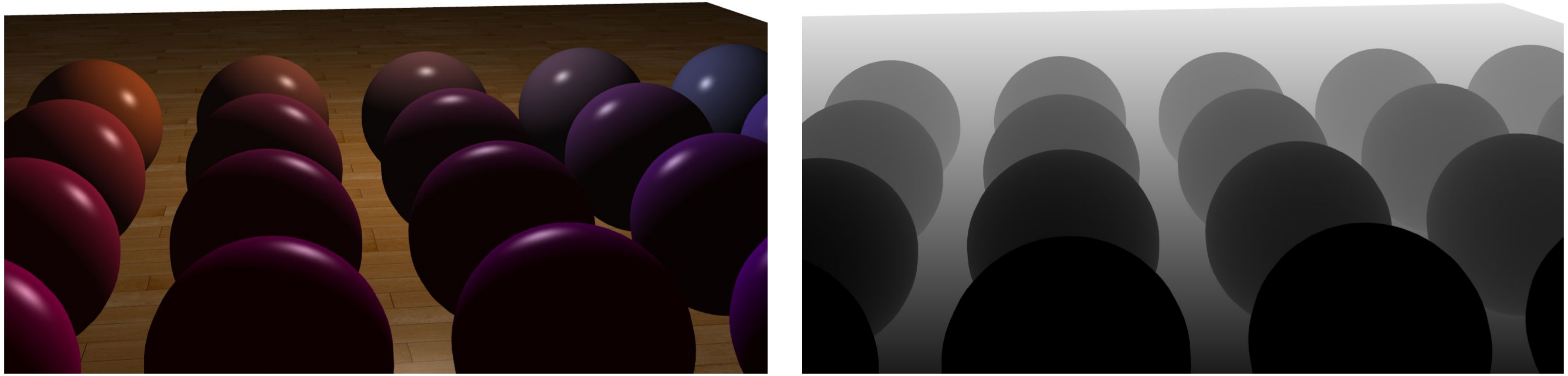
Depth testing enabled



Depth testing disabled

Depth Buffer (Z-Buffer)

- Contains normalized depth values (0-1) for all fragments
  - OpenGL tests depth value of a fragment against the content of the depth buffer
  - If test passes, the depth buffer is updated with the new depth value.
  - If depth test fails, the fragment is discarded.
- Depth test functions (various options)
  - default: `gl.depthFunc(gl.LESS);` (the smaller value wins)

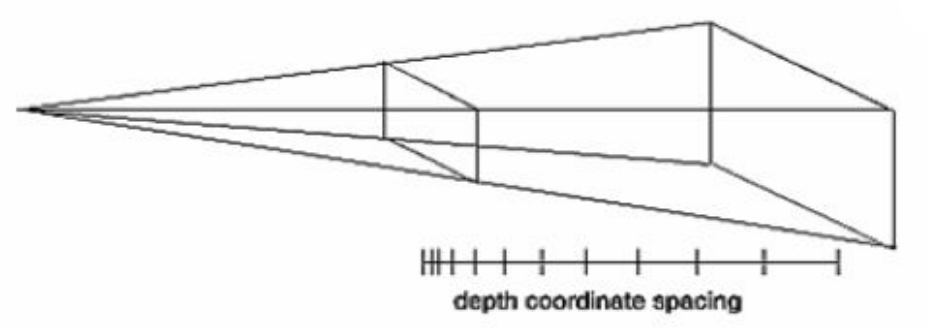


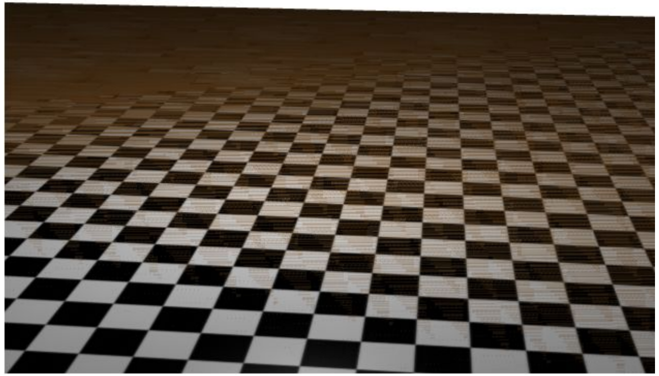
Demo: https://jku-icg.github.io/cg_demo/00_zbuffer/

see https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glDepthFunc.xhtml

Z-Fighting

- OpenGL can’t decide which face is in front
- Happens when primitives are too close together
- We can force it to happen
  - By changing the near and far clipping planes’ values of the projection
  - The closer to the near clipping plane, the denser the values get and therefore the further away, the coarser.





Blending: Alpha Values

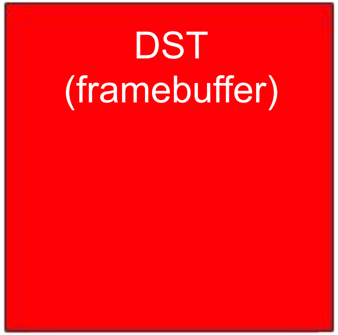
The alpha value defines an object's opacity

0 … transparent
1 … opaque
(depending on blend function)

A common blend function is:

```
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
```

(will use that in the upcoming labs)

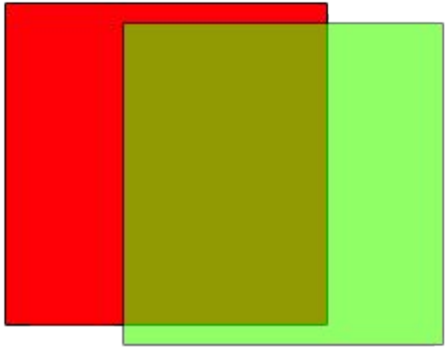


(1.0, 0.0, 0.0, 1.0)

(0.0, 1.0, 0.0, 0.6)

$$\bar{C}_{result} = \begin{pmatrix}0.0\\1.0\\0.0\\0.6\end{pmatrix} * \boxed{0.6} + \begin{pmatrix}1.0\\0.0\\0.0\\1.0\end{pmatrix} * \boxed{(1-0.6)}$$

https://learnopengl.com/Advanced-OpenGL/Blending



(0.4, 0.6, 0.0, 0.76)

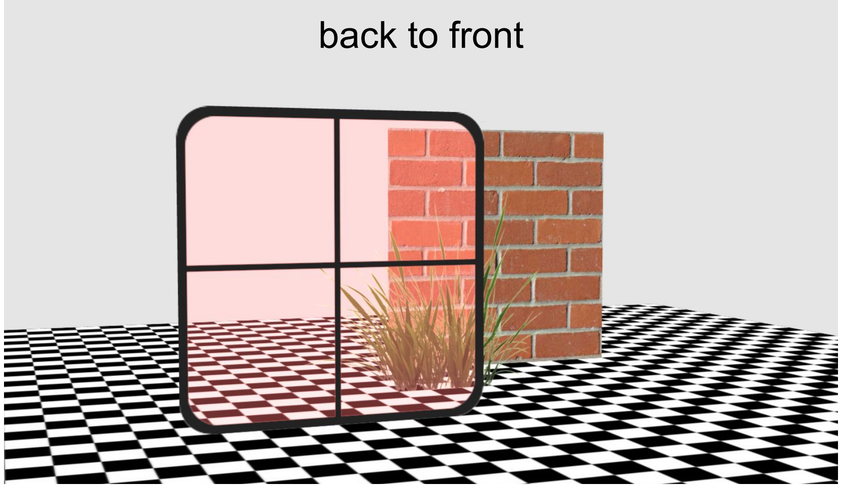
Blending: Order

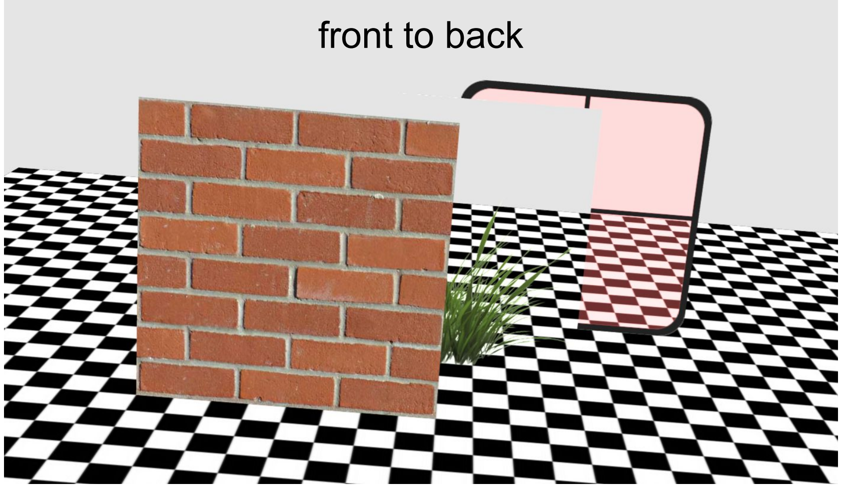
Demo: https://jku-icg.github.io/cg_demo/00_blending/

Rendering order is important (ignore it in your CG project!)

- the depth test discards fragments that would be needed
- (thus, sometimes it makes sense to disable depth testing; e.g., particles)

In practice: sort objects by depth and render from back to front


back to front


front to back

Blending: Blend Functions

OpenGL offers various blend functions

- Defining how pixels are blended, see [learnopengl.com]() or https://www.andersriggelsen.dk/glblendfunc.php
- Defines a way of blending an incoming pixel (the source) with the currently stored one (the destination):

  `gl.blendFunc(source, destination).`

Furthermore, the blend equation can be changed

- e.g., addition (very common): `gl.blendEquation(gl.FUNC_ADD);`
- Additionally, subtraction, min, max, is supported.

$$\bar{C}_{result} = \begin{pmatrix} 0.0 \\ 1.0 \\ 0.0 \\ 0.6 \end{pmatrix} * 0.6 \boxed{+} \begin{pmatrix} 1.0 \\ 0.0 \\ 0.0 \\ 1.0 \end{pmatrix} * (1 - 0.6)$$

Computer Graphics

Lab 3: Per-Fragment Operations
Tutorial

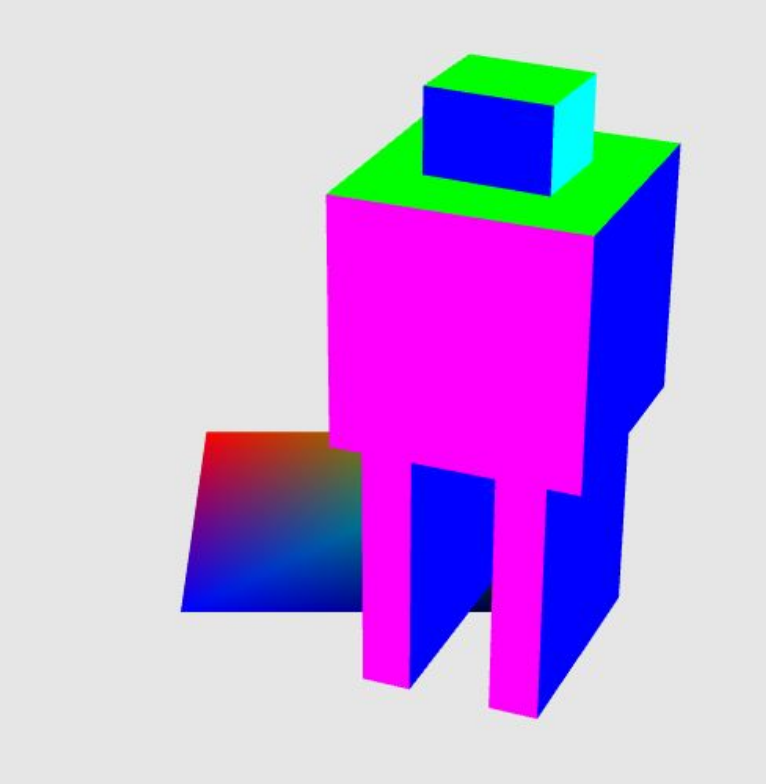
Agenda for This Week

Per-Fragment Operations

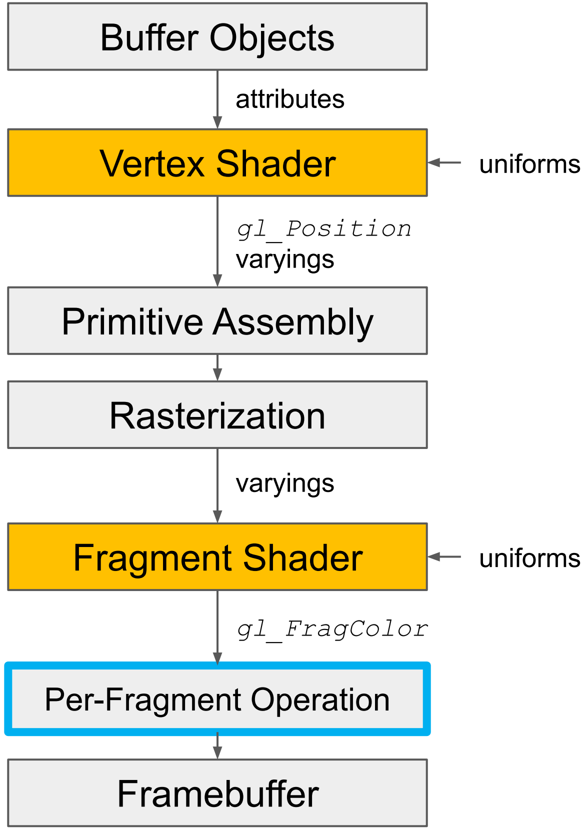
- Depth Handling
- Blending
- Tutorial (coding)

Scene Graphs

- Abstraction into Nodes
- Scene graph traversal
- Base node class
- Render nodes
- Transformation nodes
- Implement robot using a scene graph
- Tutorial (coding)



Per-Fragment Operation



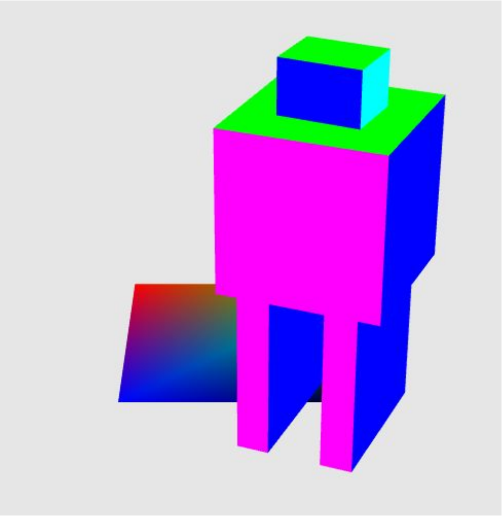
- additional tests/operations per fragment
- decides if (and how) fragments are written into the framebuffer
- examples:
  - depth test
    the closer fragment should be drawn
  - blending
    in case of semi transparent fragments
  - ...

Task 0: Depth Handling

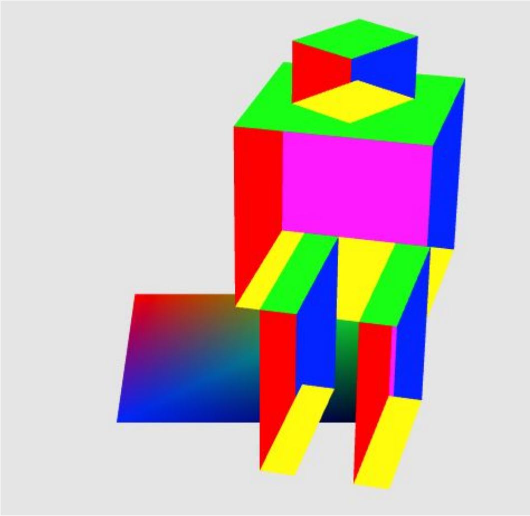
Depth testing has to be enabled to tell OpenGL to perform a depth test and use the depth buffer: `gl.enable(gl.DEPTH_TEST);`

Try to turn it by: `gl.disable(gl.DEPTH_TEST);`

Then revert it, again.

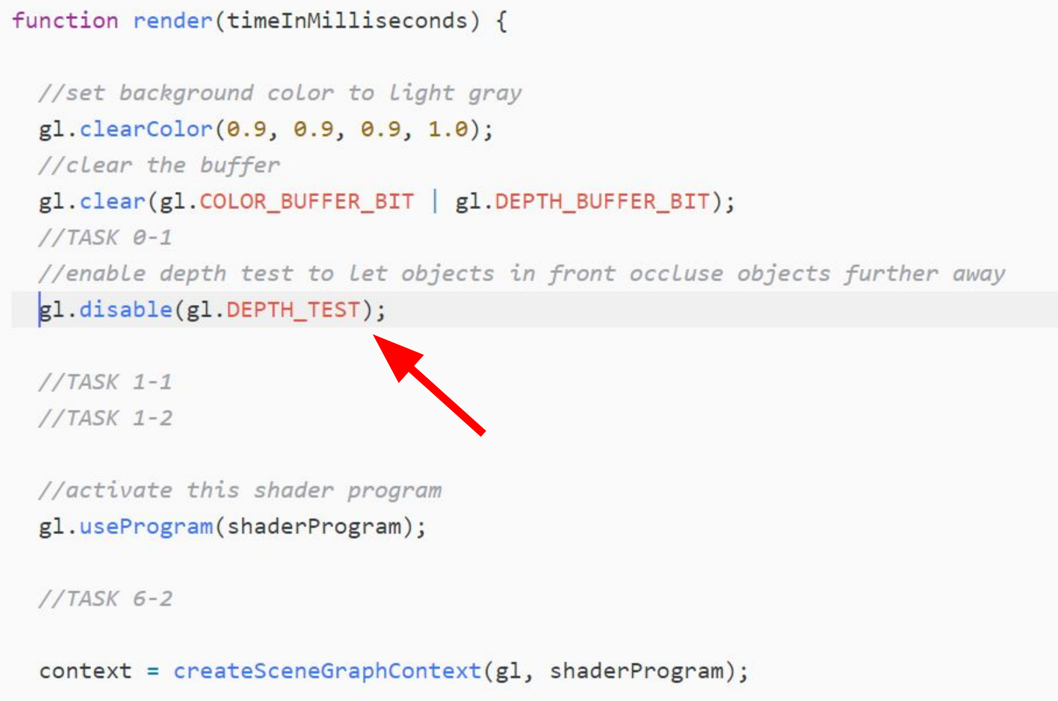


Depth testing enabled



Depth testing disabled

Task 0: Solution



```
function render(timeInMilliseconds) {

  //set background color to light gray
  gl.clearColor(0.9, 0.9, 0.9, 1.0);
  //clear the buffer
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
  //TASK 0-1
  //enable depth test to let objects in front occluse objects further away
  gl.disable(gl.DEPTH_TEST);

  //TASK 1-1
  //TASK 1-2

  //activate this shader program
  gl.useProgram(shaderProgram);

  //TASK 6-2

  context = createSceneGraphContext(gl, shaderProgram);
```

main.js

Task 1: Blending

Goal: Make the robot semi-transparent (50%)

Step 1: Enable blending by calling `gl.enable(gl.BLEND);`

Step 2: Set blend function:

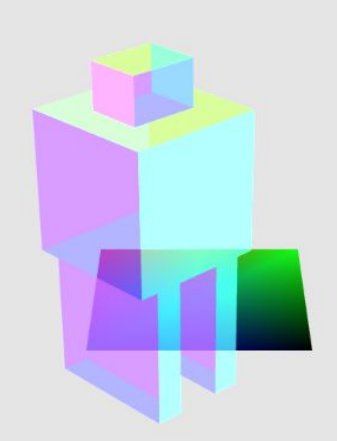
```
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
```

Step 3: Provide alpha value as uniform to fragment shader

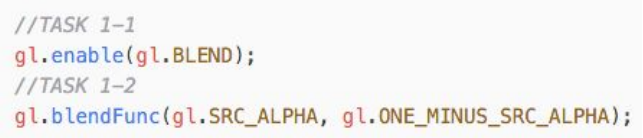
- Use `gl.uniform1f` to set `u_alpha` to **0.5** for **robot**
- Use `gl.uniform1f` to set `u_alpha` to **1.0** for **quad**
- Usage: `gl.uniform1f(location, alpha)`

Step 4: Incorporate `u_alpha` in fragment shader

- Variable is already defined in fragment shader
- Transparency is fourth component of the color vector (RGBA)



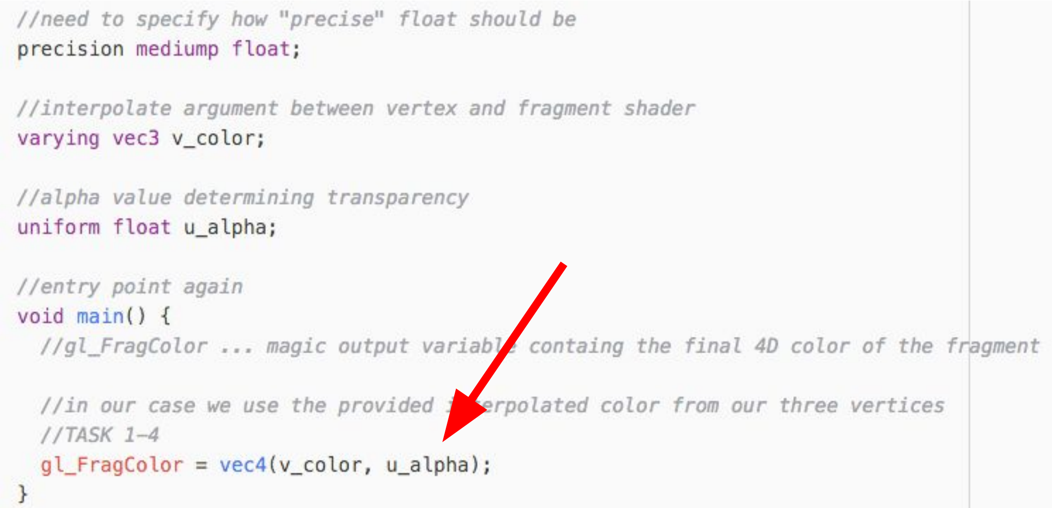
Task 1: Solution



```
//TASK 1-1
gl.enable(gl.BLEND);
//TASK 1-2
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
```

```
//set alpha value for blending
//TASK 1-3
gl.uniform1f(gl.getUniformLocation(context.shader, "u_alpha"), 0.5);
```

main.js



```
//need to specify how "precise" float should be
precision mediump float;

//interpolate argument between vertex and fragment shader
varying vec3 v_color;

//alpha value determining transparency
uniform float u_alpha;

//entry point again
void main() {
  //gl_FragColor ... magic output variable containg the final 4D color of the fragment

  //in our case we use the provided interpolated color from our three vertices
  //TASK 1-4
  gl_FragColor = vec4(v_color, u_alpha);
}
```

simple.fs.glsl

JOHANNES KEPLER
UNIVERSITY LINZ

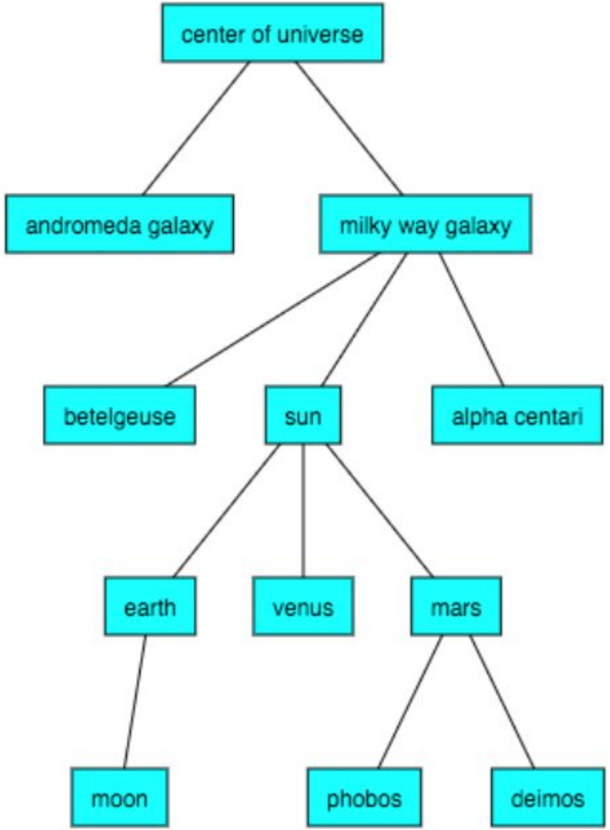
INSTITUTE OF
COMPUTER GRAPHICS

Computer Graphics

Lab 3: Scene Graphs

Agenda for This Week

- Per-Fragment Operations
  - Depth Handling
  - Blending
  - Tutorial (coding)
- Scene Graphs
  - Abstraction into Nodes
  - Scene graph traversal
  - Base node class
  - Render nodes
  - Transformation nodes
  - Implement robot using a scene graph
  - Tutorial (coding)



Scene Graphs

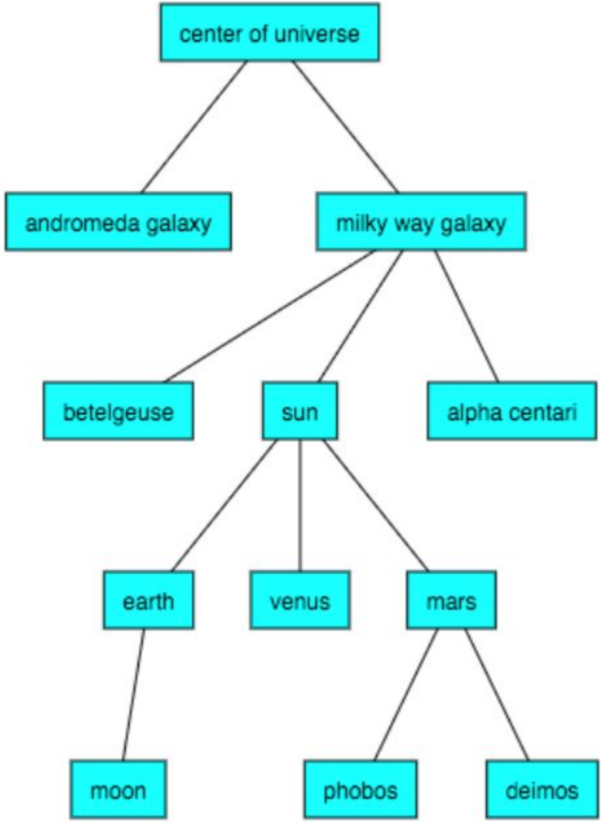
Tree-like structure (hierarchical structure) for organizing scene

Objects in the scene will be added to the graph

During rendering, the graph will be traversed recursively

Advantages:

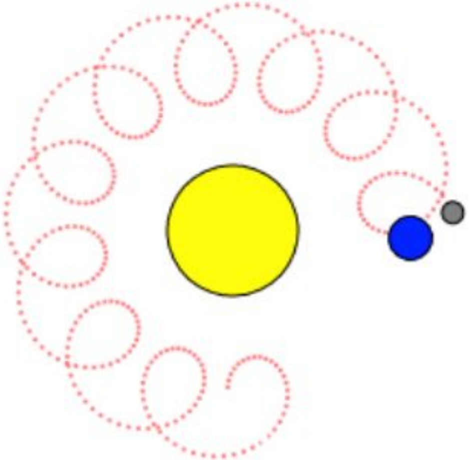
- Propagation of properties
- Unique point of access
- Reusable components
- Abstraction



http://webglfundamentals.org/webgl/lessons/webgl-scene-graph.html

Scene Graphs: Abstraction into Nodes

- Entities in graph are called nodes
  - Parent-child relationship of nodes
- Properties of a node are applied to all children
  - e.g., rotating the root node will make all sub nodes rotate too
  - In the case of transformations, each node holds a matrix and all matrices of all sub nodes are multiplied by this matrix
- Different types of nodes
  - **Geometry nodes** (sphere, models, etc.)
  - **Transformation nodes**
  - **Shader nodes** (for lighting, materials, etc.)
  - We will implement all three!



Scene Graph Traversal

Traversal works recursively

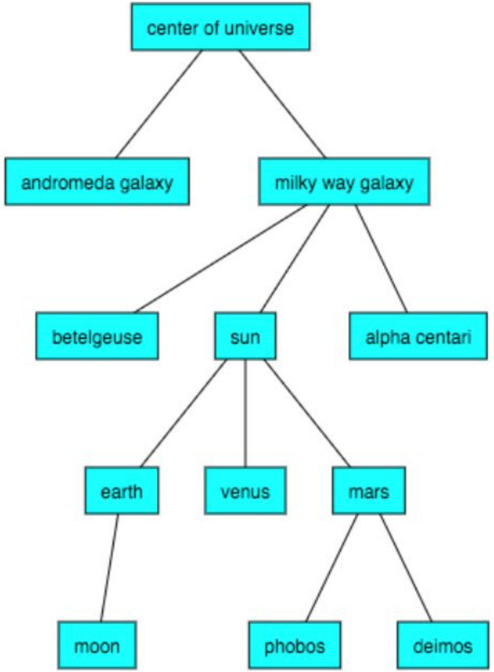
`worldMatrix = parent(worldMatrix) * self(localMatrix)`

Matrices are multiplied from top to bottom

`worldMatrix = greatGrandParent * grandParent * parent * self(localMatrix)`

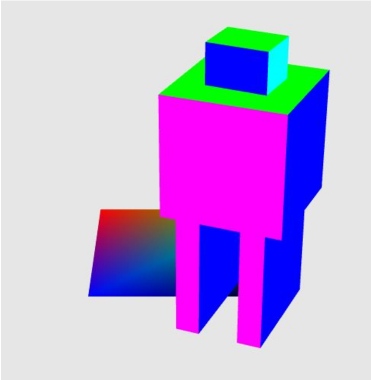
Solar system example:

`worldMatrixForMoon = galaxyMatrix * starMatrix * planetMatrix * moonMatrix;`



A Scene Graph for our Robot

- Root
  - Quad Transformation
    - Quad Render
  - Robot Transformation
    - Body Render Cube
    - Head Transformation
      - Head Render Cube
    - Left Leg Transformation
      - Left Leg Render Cube
    - Right Leg Transformation
      - Right Leg Render Cube



A Basic Node

Needs the functionality to:

- Append a child node
- Remove a child node
- Render node and all its children (recursively)

Also: a Render context

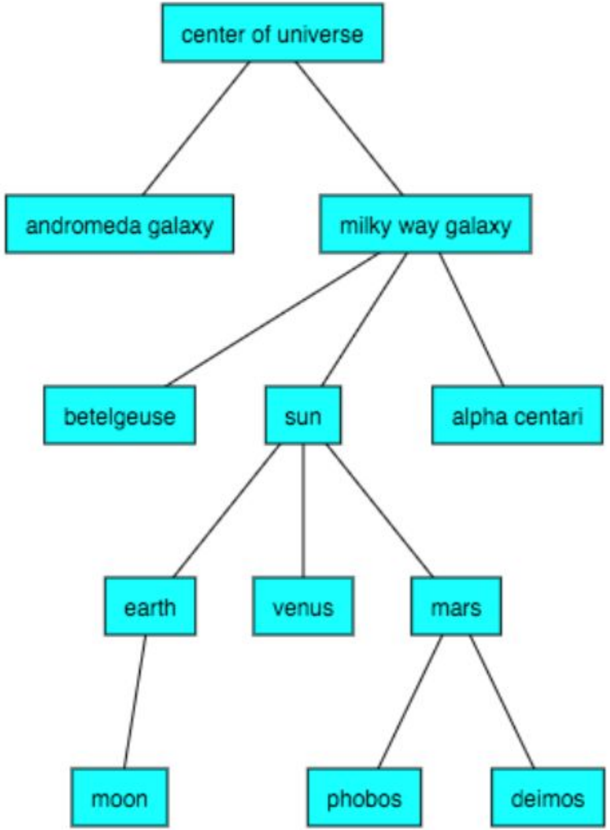
- Stores scene matrix, view matrix, projection matrix and shader
- Gives us access to shader and matrices from inside nodes

Computer Graphics

Lab 3: Scene Graphs
Tutorial

Agenda for This Week

- Per-Fragment Operations
  - Depth Handling
  - Blending
  - Tutorial (coding)
- Scene Graphs
  - Abstraction into Nodes
  - Scene graph traversal
  - Base node class
  - Render nodes
  - Transformation nodes
  - Implement robot using a scene graph
  - Tutorial (coding)

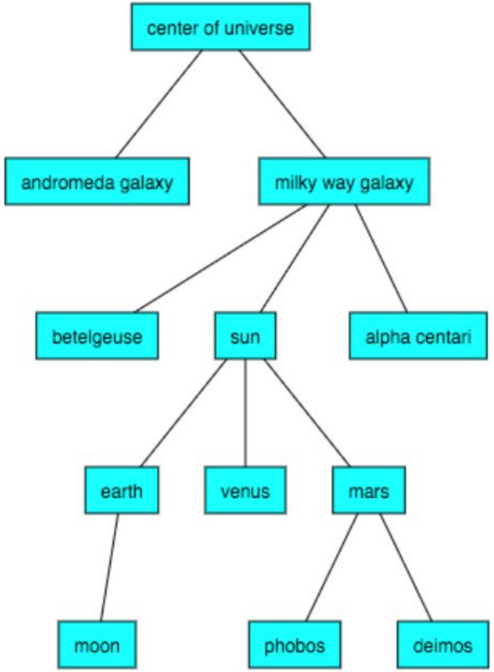


Scene Graphs

Tree-like structure (hierarchical structure)

Advantages:

- Propagation of properties
- Unique point of access
- Reusable components
- Abstraction



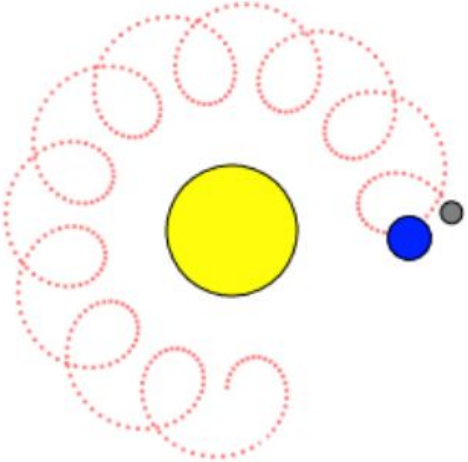
Creating a Node

We need the functionality to:

- Append a child node
- Remove a child node
- Render node and all its children (recursively)

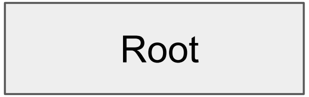
What we also need: Render context

- Stores scene matrix, view matrix, projection matrix, and shader
- Access to shader and matrices inside nodes



Base Node

```
//create scenegraph
rootNode = new SceneGraphNode();
```

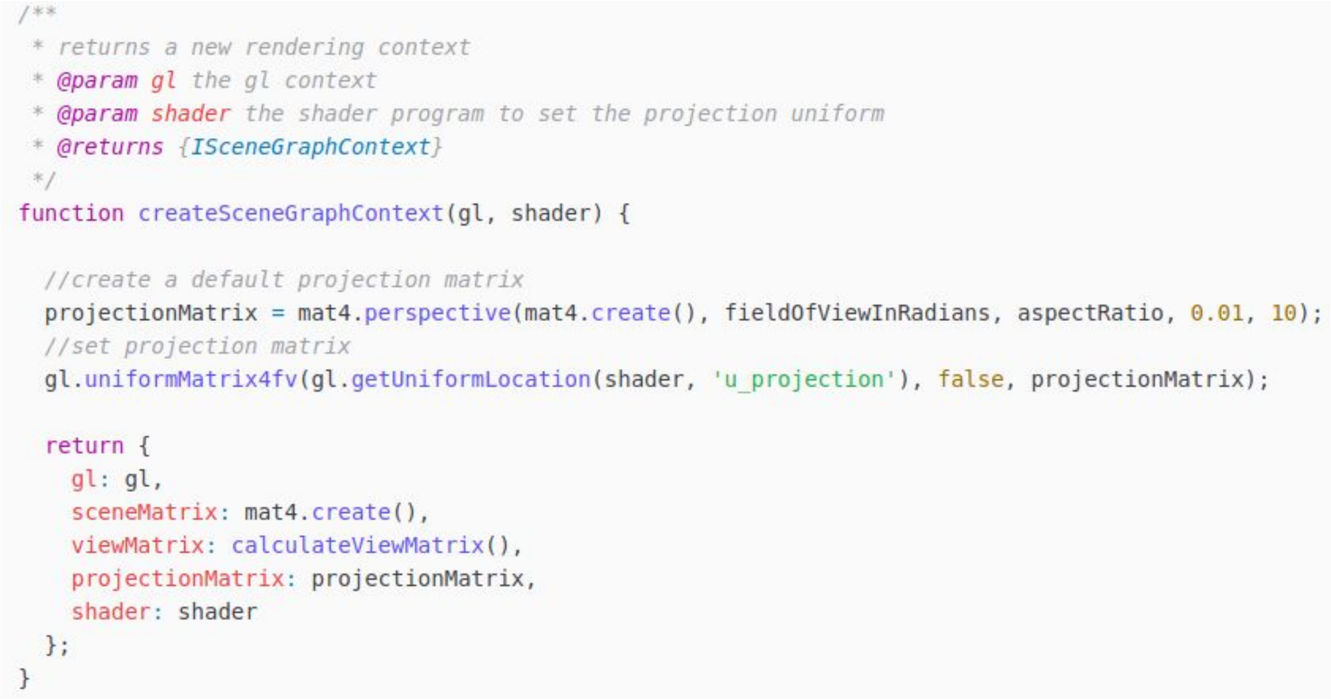


```
/**
 * base node of the scenegraph
 */
class SceneGraphNode {

  constructor() {
    this.children = [];
  }

  /**
   * appends a new child to this node
   * @param child the child to append
   * @returns {SceneGraphNode} the child
   */
  append(child) {
    this.children.push(child);
    return child;
  }

  /**
   * removes a child from this node
   * @param child
   * @returns {boolean} whether the operation was successful
   */
  remove(child) {
    var i = this.children.indexOf(child);
    if (i >= 0) {
      this.children.splice(i, 1);
    }
    return i >= 0;
  };

  /**
   * render method to render this scengraph
   * @param context
   */
  render(context) {

    //render all children
    this.children.forEach(function (c) {
      return c.render(context);
    });
  };
}
```

main.js

Render Context

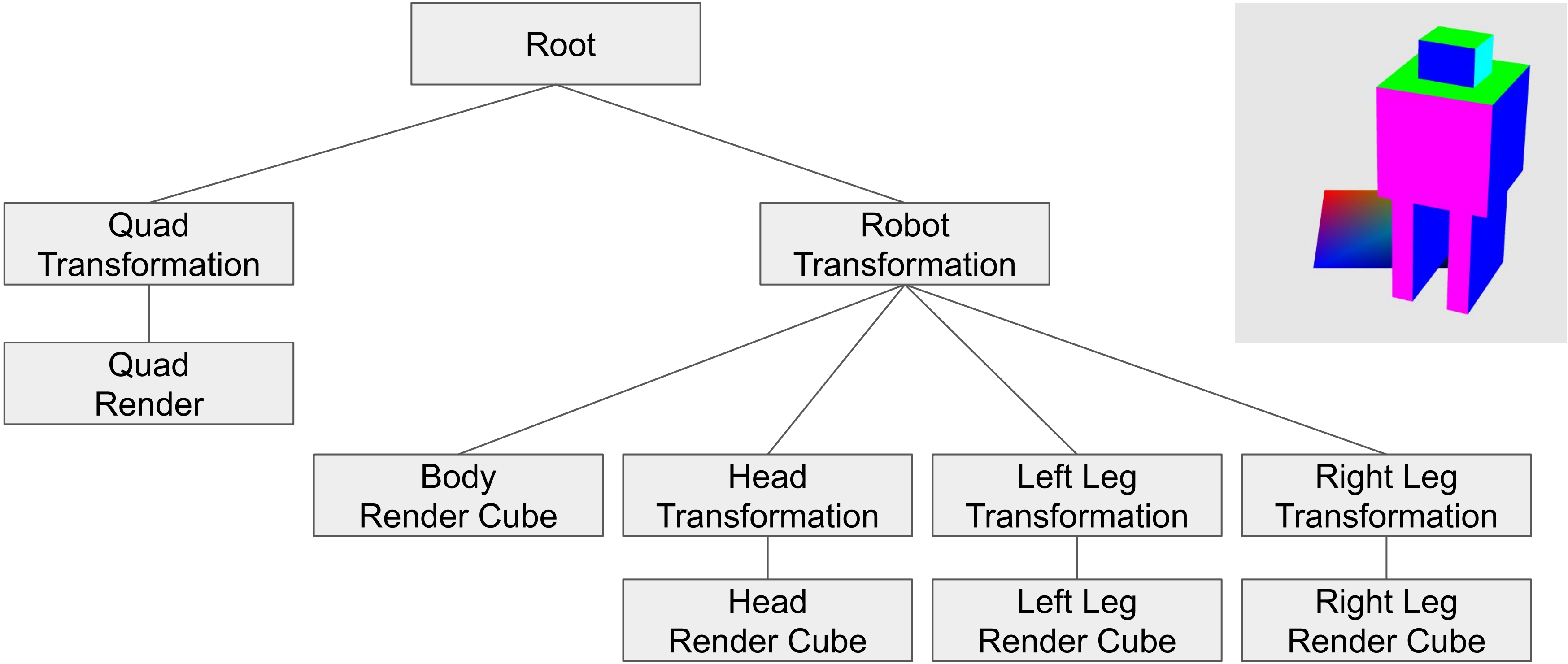
```
const context = createSceneGraphContext(gl, shaderProgram);
```



```
/**
 * returns a new rendering context
 * @param gl the gl context
 * @param shader the shader program to set the projection uniform
 * @returns {ISceneGraphContext}
 */
function createSceneGraphContext(gl, shader) {

  //create a default projection matrix
  projectionMatrix = mat4.perspective(mat4.create(), fieldOfViewInRadians, aspectRatio, 0.01, 10);
  //set projection matrix
  gl.uniformMatrix4fv(gl.getUniformLocation(shader, 'u_projection'), false, projectionMatrix);

  return {
    gl: gl,
    sceneMatrix: mat4.create(),
    viewMatrix: calculateViewMatrix(),
    projectionMatrix: projectionMatrix,
    shader: shader
  };
}
```

main.js

Let’s Make a Scene Graph for our Robot



Root

Quad
Transformation

Quad
Render

Robot
Transformation

Body
Render Cube

Head
Transformation

Head
Render Cube

Left Leg
Transformation

Left Leg
Render Cube

Right Leg
Transformation

Right Leg
Render Cube

Task 2: Create a Quad Render Node

Goal: Implement empty `QuadRenderNode` template

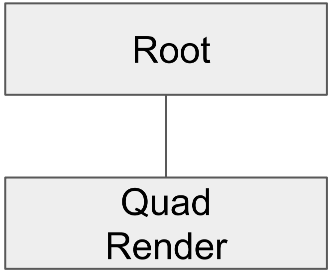
Step 0: Comment out `renderQuad` and `renderRobot` calls

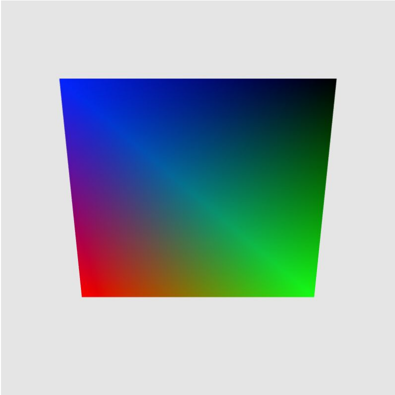
Step 1: Move quad render code to `QuadRenderNode` class

- Note: Take `sceneMatrix` and `viewMatrix` from render context (`context`)
- Note: Without transformations!

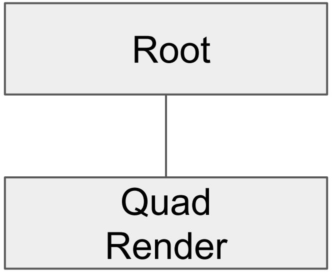
Step 2: Create node and add it to the scene graph

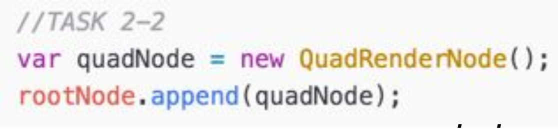
- Note: Quad looks distorted because of perspective projection and camera





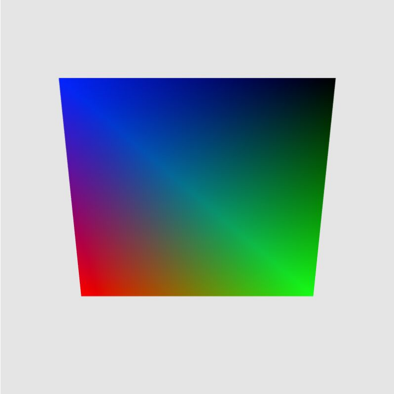
Task 2: Solution





```
//TASK 2-2
var quadNode = new QuadRenderNode();
rootNode.append(quadNode);
```

main.js



```
class QuadRenderNode extends SceneGraphNode {

  render(context) {

    //TASK 2-1
    //setting the model view and projection for the shader
    setUpModelViewMatrix(context.sceneMatrix, context.viewMatrix);
    gl.uniformMatrix4fv( gl.getUniformLocation(context.shader, 'u_projection'),
                         false, context.projectionMatrix);

    var positionLocation = gl.getAttribLocation(context.shader, 'a_position');
    gl.bindBuffer(gl.ARRAY_BUFFER, quadVertexBuffer);
    gl.vertexAttribPointer(positionLocation, 2, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(positionLocation);

    var colorLocation = gl.getAttribLocation(context.shader, 'a_color');
    gl.bindBuffer(gl.ARRAY_BUFFER, quadColorBuffer);
    gl.vertexAttribPointer(colorLocation, 4, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(colorLocation);

    //set alpha value for blending
    //TASK 1-3
    gl.uniform1f(gl.getUniformLocation(context.shader, 'u_alpha'), 1);

    // draw the bound data as 6 vertices = 2 triangles starting at index 0
    gl.drawArrays(gl.TRIANGLES, 0, 6);

    //render children
    super.render(context);
  }
}
```

main.js

Transformation Node

Stores transformation matrix

During rendering:

1. Backup current `sceneMatrix` from context
2. Multiply `sceneMatrix` with local matrix
3. Render children
4. Restore previous `sceneMatrix`

```
//TASK 3-0
/**
 * a transformation node, i.e applied a transformation matrix to its successors
 */
class TransformationSceneGraphNode extends SceneGraphNode {
  /**
   * the matrix to apply
   * @param matrix
   */
  constructor(matrix) {
    super();
    this.matrix = matrix || mat4.create();
  }

  render(context) {
    //backup previous one
    var previous = context.sceneMatrix;
    //set current world matrix by multiplying it
    if (previous === null) {
      context.sceneMatrix = mat4.clone(this.matrix);
    }
    else {
      context.sceneMatrix = mat4.multiply(mat4.create(), previous, this.matrix);
    }

    //render children
    super.render(context);
    //restore backup
    context.sceneMatrix = previous;
  }

  setMatrix(matrix) {
    this.matrix = matrix;
  }
}
```

main.js

Task 3: Create a Transformation Node

Goal: Apply quad transformations

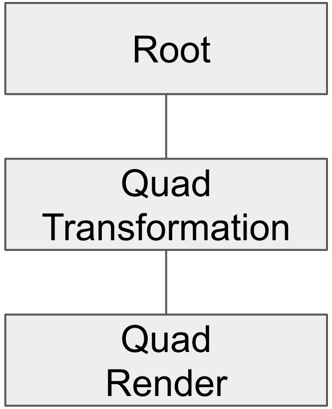
Step 0: Take a look at `TransformationSceneGraphNode`

Step 1: Create local transformation matrix

 Move transformation code from `renderQuad`

Step 2: Create transformation node and add to scene graph

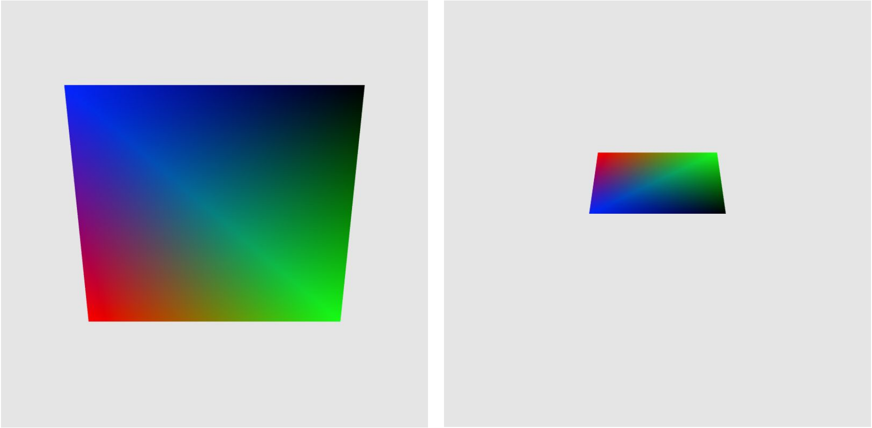
 Hint: Don’t forget to update existing scene graph parent-child relationships!



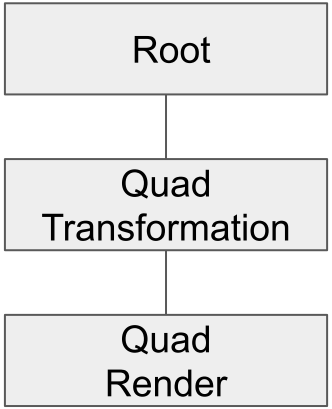
Root

Quad Transformation

Quad Render



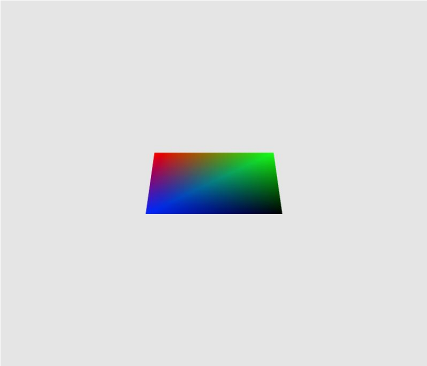
Task 3: Solution

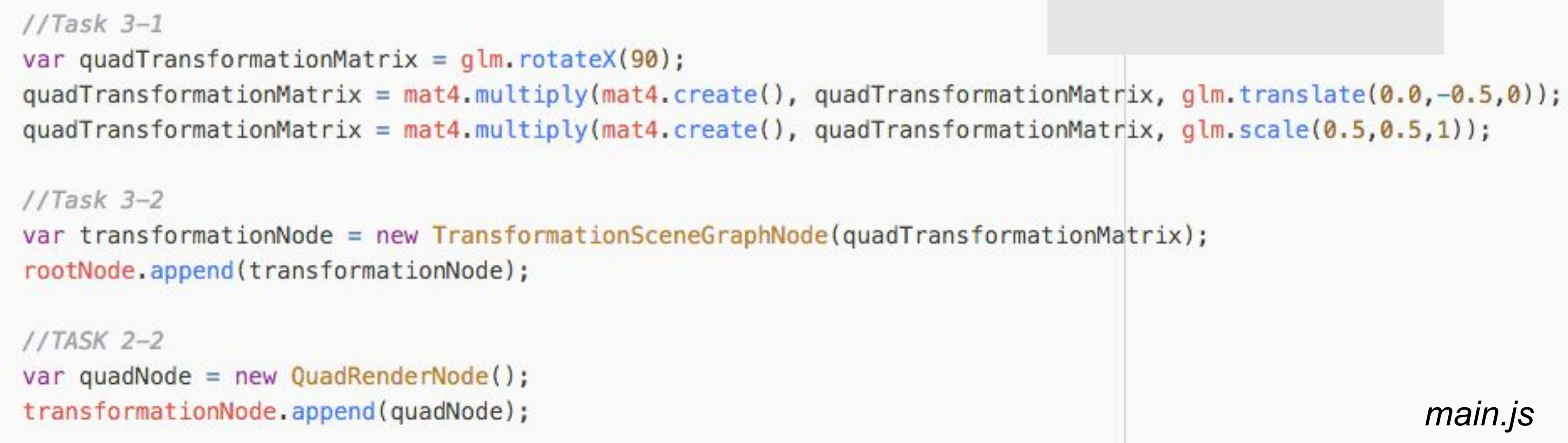


Root

Quad Transformation

Quad Render





```
//Task 3-1
var quadTransformationMatrix = glm.rotateX(90);
quadTransformationMatrix = mat4.multiply(mat4.create(), quadTransformationMatrix, glm.translate(0.0,-0.5,0));
quadTransformationMatrix = mat4.multiply(mat4.create(), quadTransformationMatrix, glm.scale(0.5,0.5,1));

//Task 3-2
var transformationNode = new TransformationSceneGraphNode(quadTransformationMatrix);
rootNode.append(transformationNode);

//TASK 2-2
var quadNode = new QuadRenderNode();
transformationNode.append(quadNode);
```

main.js

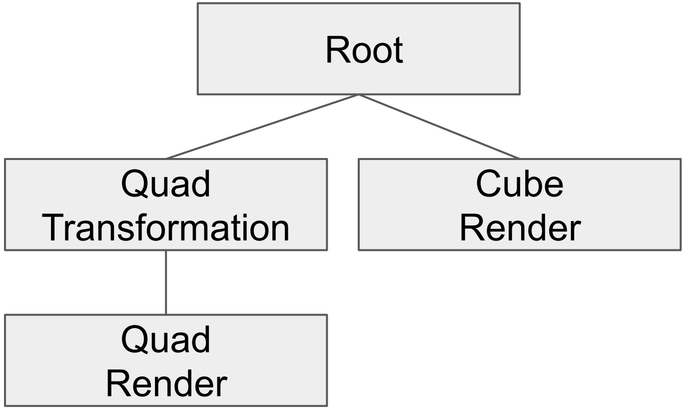
Task 4: Create a Cube Render Node

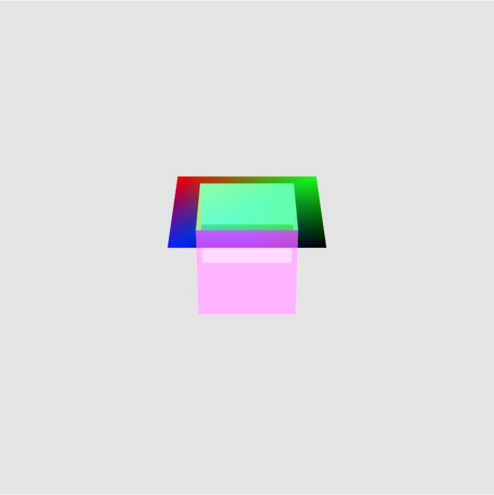
Goal: Render a cube

Step 1: Implement `CubeRenderNode`

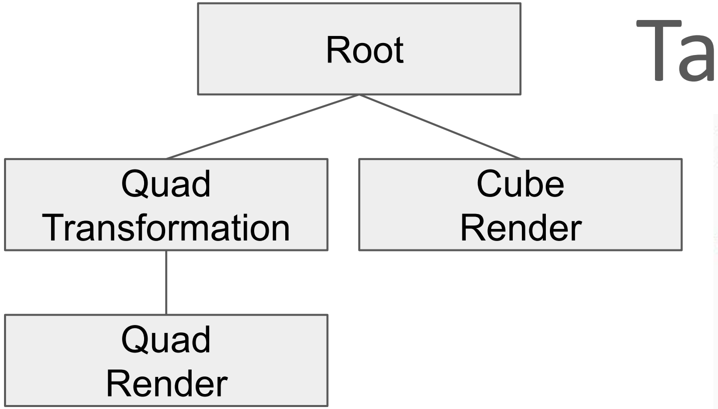
- Take `QuadRenderNode` as template
- Use cube render code from `renderRobot`

Step 2: Create cube node and add as child of root node





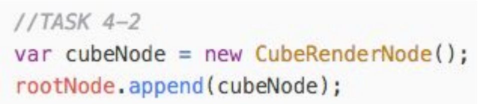
Task 4: Solution

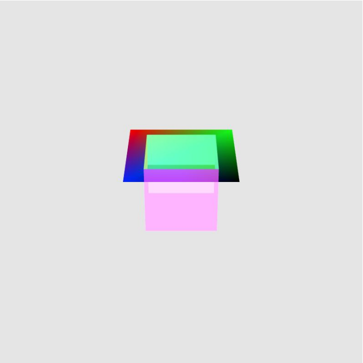

Root

Quad Transformation

Cube Render

Quad Render


```
//TASK 4-2
var cubeNode = new CubeRenderNode();
rootNode.append(cubeNode);
```




```
//TASK 4-1
/**
 * a cube node that renders a cube at its local origin
 */
class CubeRenderNode extends SceneGraphNode {
  render(context) {
    //setting the model view and projection for the shader
    setUpModelViewMatrix(context.sceneMatrix, context.viewMatrix);
    gl.uniformMatrix4fv(  gl.getUniformLocation(context.shader, 'u_projection'),
                          false, context.projectionMatrix);

    var positionLocation = gl.getAttribLocation(context.shader, 'a_position');
    gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexBuffer);
    gl.vertexAttribPointer(positionLocation, 3, gl.FLOAT, false,0,0) ;
    gl.enableVertexAttribArray(positionLocation);

    var colorLocation = gl.getAttribLocation(context.shader, 'a_color');
    gl.bindBuffer(gl.ARRAY_BUFFER, cubeColorBuffer);
    gl.vertexAttribPointer(colorLocation, 3, gl.FLOAT, false,0,0) ;
    gl.enableVertexAttribArray(colorLocation);

    //set alpha value for blending
    //TASK 1-3
    gl.uniform1f(gl.getUniformLocation(context.shader, 'u_alpha'), 0.5);

    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, cubeIndexBuffer);
    gl.drawElements(gl.TRIANGLES, cubeIndices.length, gl.UNSIGNED_SHORT, 0);

    //render children
    super.render(context);
  }
}
```

main.js

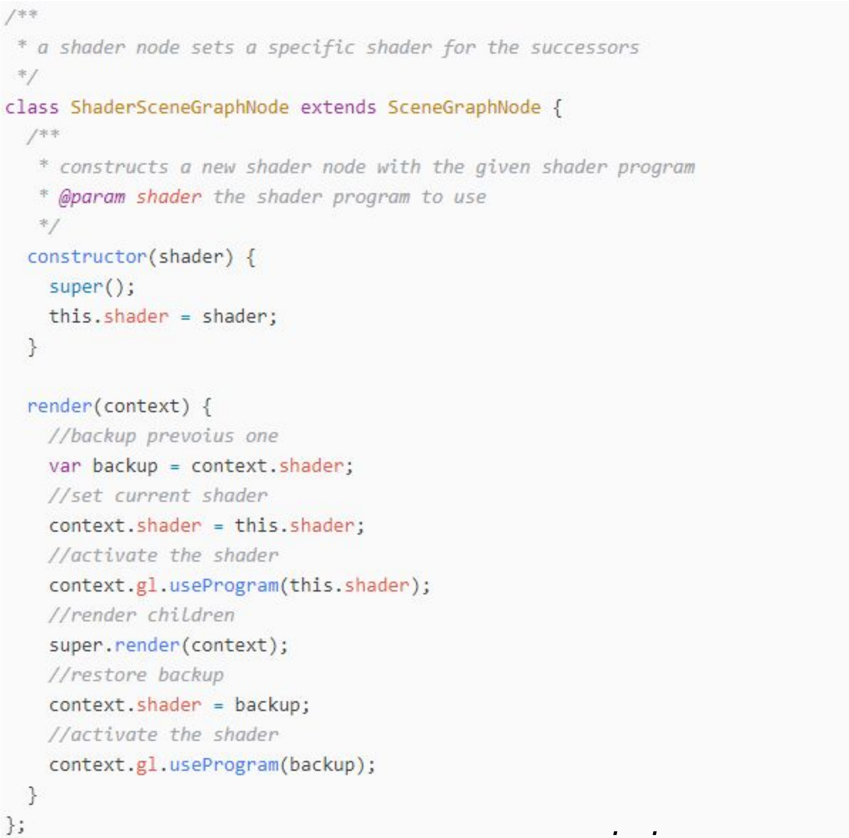
Shader Node

Holds shader program as variable

Allows us to use multiple shaders

During rendering:

1. Backup current `shader` from context
2. Use program (and set projection)
3. Render children
4. Restore previous `shader`



```
/**
 * a shader node sets a specific shader for the successors
 */
class ShaderSceneGraphNode extends SceneGraphNode {
  /**
   * constructs a new shader node with the given shader program
   * @param shader the shader program to use
   */
  constructor(shader) {
    super();
    this.shader = shader;
  }

  render(context) {
    //backup prevoius one
    var backup = context.shader;
    //set current shader
    context.shader = this.shader;
    //activate the shader
    context.gl.useProgram(this.shader);
    //render children
    super.render(context);
    //restore backup
    context.shader = backup;
    //activate the shader
    context.gl.useProgram(backup);
  }
};
```

main.js

Task 5: Adding a Shader Node

Goal: Assign yellow color to floor quad

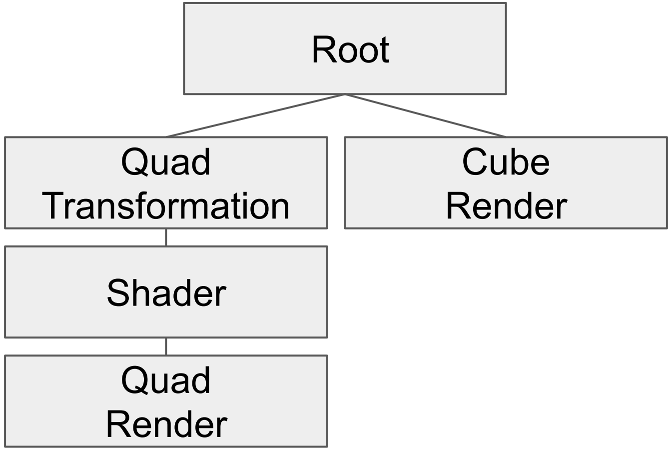
Step 1: Duplicate `simple.vs.glsl` shader and rename it

Step 2: Modify shader to apply static yellow color (R:1, G:1, B:0)

Step 3: Load new vertex shader as resource in `loadResources`

Step 4: Create new shader node and add it before the quad node

- Note: create new shader program and pass it to constructor of shader node



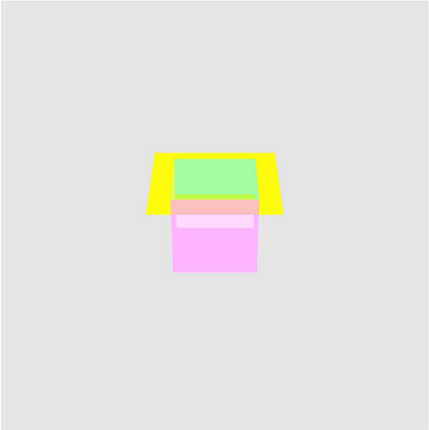
Root

Quad Transformation

Cube Render

Shader

Quad Render

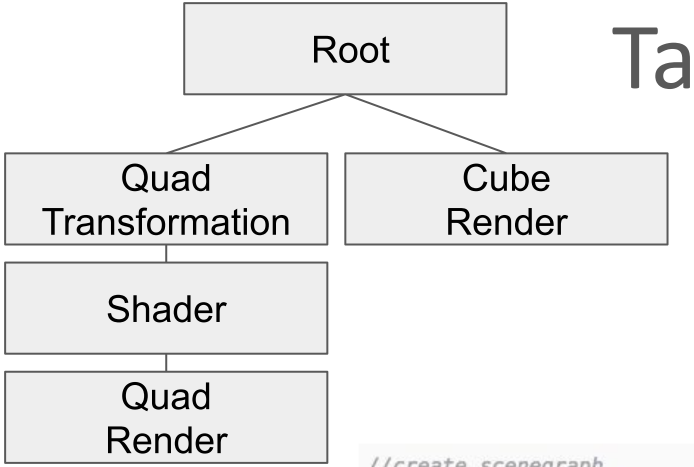


Task 5-1 and 5-2: Solution

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

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

  //TASK 5-2
  //we don't use a_color anymore
  a_color;

  //setting a static color (yellow) to the output varying color
  v_color = vec3(1,1,0);
}
```

static_color.vs.glsl

Task 5-3 and 5-4: Solution


Root

Quad Transformation

Cube Render

Shader

Quad Render


```
//load the shader resources using a utility function
loadResources({
  vs: 'shader/simple.vs.glsl',
  fs: 'shader/simple.fs.glsl',
  //TASK 5-3
  staticcolorvs: 'shader/static_color.vs.glsl'
}).then(function (resources /*an object containing our keys with the loaded resources*/) {
  init(resources);

  //render one frame
  render();
});
```

main.js


```
//create scenegraph
rootNode = new SceneGraphNode();

//Task 3-1
var quadTransformationMatrix = glm.rotateX(90);
quadTransformationMatrix = mat4.multiply(mat4.create(), quadTransformationMatrix, glm.translate(0.0,-0.5,0));
quadTransformationMatrix = mat4.multiply(mat4.create(), quadTransformationMatrix, glm.scale(0.5,0.5,1));

//Task 3-2
var transformationNode = new TransformationSceneGraphNode(quadTransformationMatrix);
rootNode.append(transformationNode);

//TASK 5-3
var staticColorShaderNode = new ShaderSceneGraphNode(createProgram(gl, resources.staticcolorvs, resources.fs));
transformationNode.append(staticColorShaderNode);

//TASK 2-2
var quadNode = new QuadRenderNode();
staticColorShaderNode.append(quadNode);
```

main.js

Reference Scene Graph Implementation

Frameworks typically provide a scene graph implementation

Example: Three.js

Our framework provides the following (and more) nodes:

- `SGNode` (base class)
- `TransformationSGNode`
- `ShaderSGNode`
- `RenderSGNode`

In upcoming labs, we will use the framework's scene graph implementation

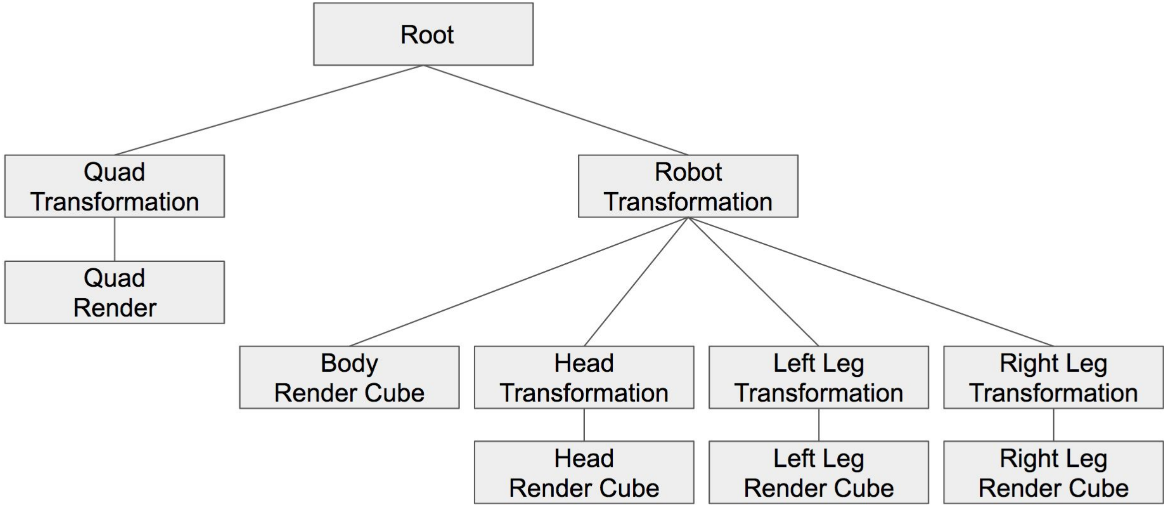
Use framework nodes in your CG project!

Task 6: Create Robot (At Home)

Goal: Rebuild robot from last lab as a scene graph

Step 1: Construct a robot scene graph in `createRobot` function

Step 2: Update rotation in `render` function



Root

Quad Transformation

Quad Render

Robot Transformation

Body Render Cube

Head Transformation

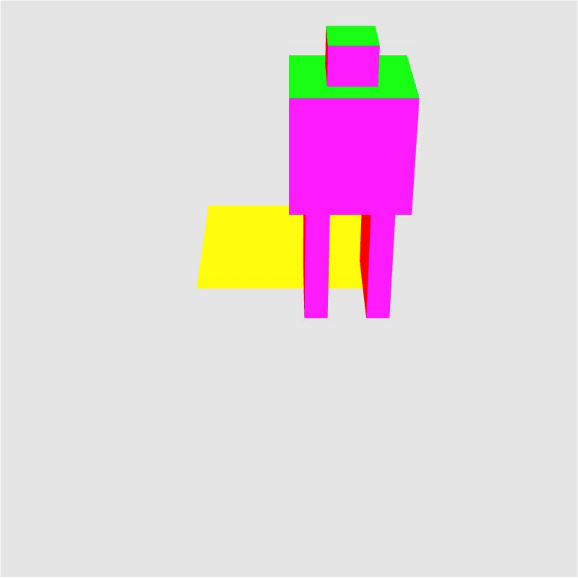
Head Render Cube

Left Leg Transformation

Left Leg Render Cube

Right Leg Transformation

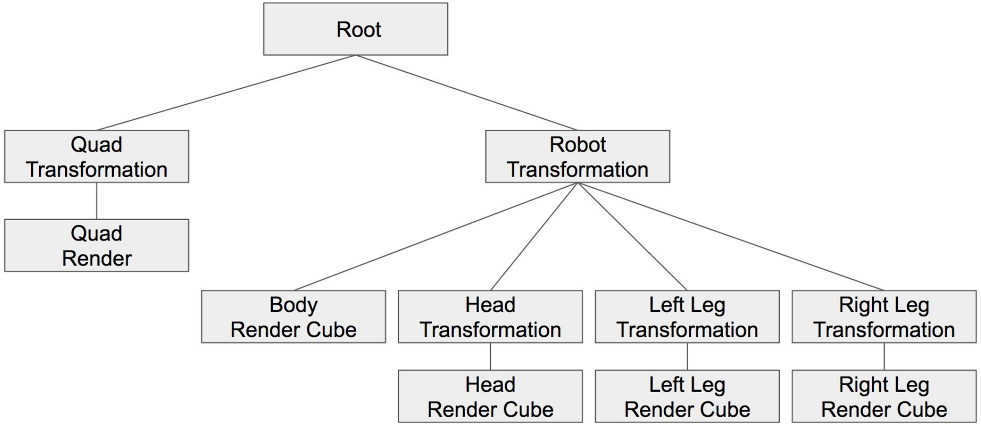
Right Leg Render Cube



Task 6-1: Solution

```
function createRobot(rootNode) {

  //TASK 6

  //transformations of whole body
  var robotTransformationMatrix = mat4.multiply(mat4.create(), mat4.create(), glm.rotateY(animatedAngle/2));
  robotTransformationMatrix = mat4.multiply(mat4.create(), robotTransformationMatrix, glm.translate(0.3,0.9,0));
  robotTransformationNode = new TransformationSceneGraphNode(robotTransformationMatrix);
  rootNode.append(robotTransformationNode);

  //body
  cubeNode = new CubeRenderNode();
  robotTransformationNode.append(cubeNode);

  //transformation of head
  var headTransformationMatrix = mat4.multiply(mat4.create(), mat4.create(), glm.rotateY(animatedAngle));
  headTransformationMatrix = mat4.multiply(mat4.create(), headTransformationMatrix, glm.translate(0.0,0.4,0));
  headTransformationMatrix = mat4.multiply(mat4.create(), headTransformationMatrix, glm.scale(0.4,0.33,0.5));
  headTransformationNode = new TransformationSceneGraphNode(headTransformationMatrix);
  robotTransformationNode.append(headTransformationNode);

  //head
  cubeNode = new CubeRenderNode();
  headTransformationNode.append(cubeNode);

  //transformation of left leg
  var leftLegTransformationMatrix = mat4.multiply(mat4.create(), mat4.create(), glm.translate(0.16,-0.6,0));
  leftLegTransformationMatrix = mat4.multiply(mat4.create(), leftLegTransformationMatrix, glm.scale(0.2,1,1));
  var leftLegTransformationNode = new TransformationSceneGraphNode(leftLegTransformationMatrix);
  robotTransformationNode.append(leftLegTransformationNode);

  //left leg
  cubeNode = new CubeRenderNode();
  leftLegTransformationNode.append(cubeNode);

  //transformation of right leg
  var rightLegTransformationMatrix = mat4.multiply(mat4.create(), mat4.create(), glm.translate(-0.16,-0.6,0));
  rightLegTransformationMatrix = mat4.multiply(mat4.create(), rightLegTransformationMatrix, glm.scale(0.2,1,1));
  var rightLegtTransformationNode = new TransformationSceneGraphNode(rightLegTransformationMatrix);
  robotTransformationNode.append(rightLegtTransformationNode);

  //right leg
  cubeNode = new CubeRenderNode();
  rightLegtTransformationNode.append(cubeNode);
}
```

main.js



Root

Quad Transformation

Quad Render

Robot Transformation

Body Render Cube

Head Transformation

Head Render Cube

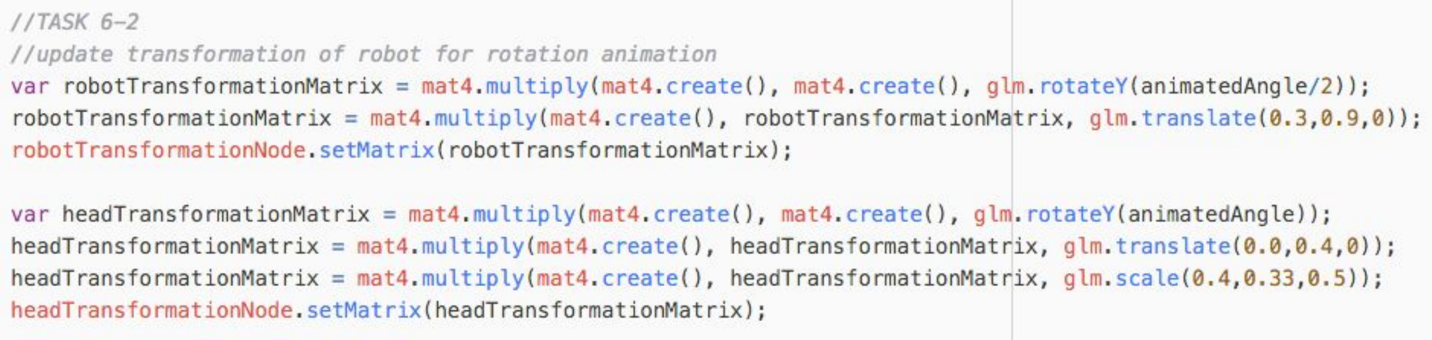
Left Leg Transformation

Left Leg Render Cube

Right Leg Transformation

Right Leg Render Cube

Task 6-2: Solution



```
//TASK 6-2
//update transformation of robot for rotation animation
var robotTransformationMatrix = mat4.multiply(mat4.create(), mat4.create(), glm.rotateY(animatedAngle/2));
robotTransformationMatrix = mat4.multiply(mat4.create(), robotTransformationMatrix, glm.translate(0.3,0.9,0));
robotTransformationNode.setMatrix(robotTransformationMatrix);

var headTransformationMatrix = mat4.multiply(mat4.create(), mat4.create(), glm.rotateY(animatedAngle));
headTransformationMatrix = mat4.multiply(mat4.create(), headTransformationMatrix, glm.translate(0.0,0.4,0));
headTransformationMatrix = mat4.multiply(mat4.create(), headTransformationMatrix, glm.scale(0.4,0.33,0.5));
headTransformationNode.setMatrix(headTransformationMatrix);
```

main.js - in render function

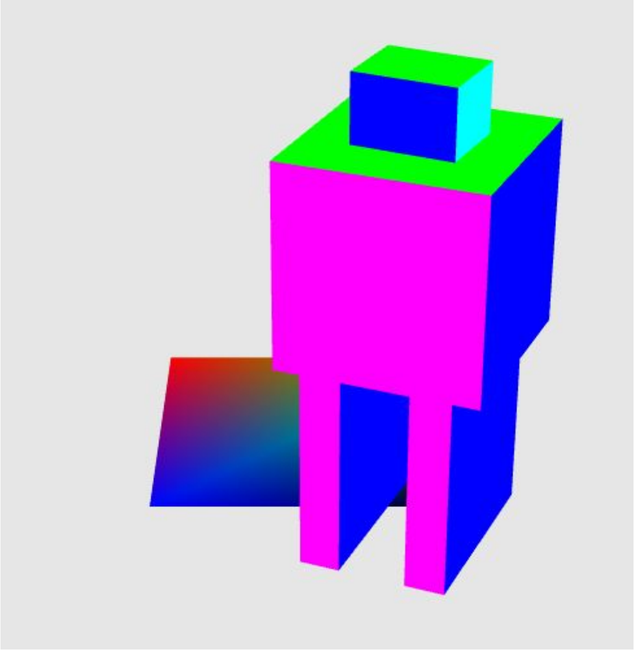
Recap

Per-Fragment Operations

- Depth Handling
- Blending
- Tutorial (coding)

Scene Graphs

- Abstraction into Nodes
- Scene graph traversal
- Base node class
- Render nodes
- Transformation nodes
- Implement robot using a scene graph
- Tutorial (coding)



Next Time

Illumination and shading

- Light
- Material