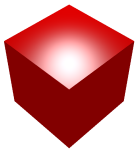
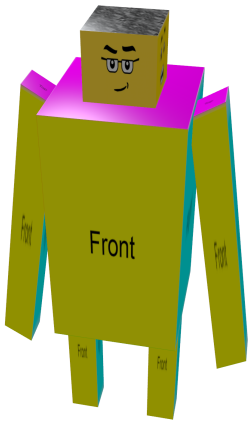
Computer Graphics Tick 2

Scene Graphs with OpenGL

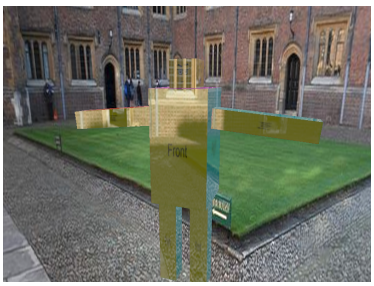
(a) A simple cube



(b) A shaded and transformed cube



(c) Animated robot



(d) Robot in Cambridge

1 Introduction

In this Tick you will write code for organising and animating a 3D scene using a technique called *Scene Graph*. You will then render this scene with OpenGL using *rasterization*.

The document starts with an instruction on how to find, compile and run the skeleton code from the command line. We recommend that you follow the "Working with IDEs" document and set up your development in an IDE. Section 3 of this instruction will give you an overview of the skeleton code so that you can understand how OpenGL applications are structured. If you have some experience with OpenGL, you can jump directly into Section 4, which describes the tasks to be completed.

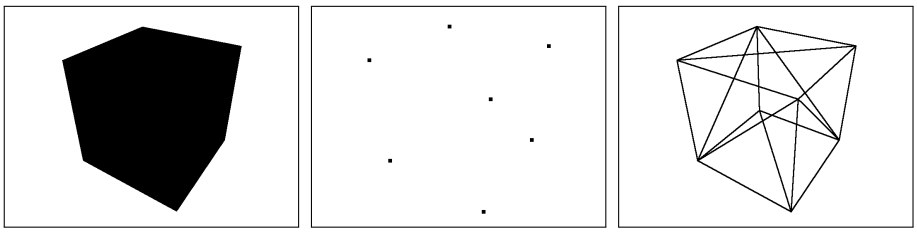


Figure 2: If you have set Tick 2 up correctly, you will see a black cube. Press P to see the vertices, and W to see that the cube is made up of triangles.

2 Getting started

The skeleton code can be found on Moodle course page. It contains the following files and directories:

- tick2
  - lib
    - JOML — *Java maths library for OpenGL calculations*
    - lwjgl — *Lightweight Java Game Library*
  - resources
    - `cube_fragment_shader.glsl` — *Processes fragments created by the rasterizer*
    - `cube_vertex_shader.glsl` — *Processes vertex data*
    - skybox_vertext_shader.glsl — *Processes skybox vertex data*
    - `skybox_fragment_shader.glsl` — *Processes fragments of skybox*
    - cubemap.png — *Texture image for your cube*
  - src/gfx/tick2
    - Camera.java — *Controls 3D camera position*
    - OpenGLApplication.java — *Uses OpenGL to interface with the GPU*
    - Shader.java — *Compiles a shader – a program for the GPU*
    - ShaderProgram.java — *Combines the vertex and fragment shaders together*
    - Texture.java — *Creates OpenGL texture object from an image file*
    - Tick2.java — *Main class*
    - Mesh.java — *Abstract class encapsulating a 3D mesh object*
    - CubeMesh.java — *Mesh subclass that defines a Cube mesh*
    - `CubeRobot.java` — *A scene object that will represent a robot made of cubes*
    - Skybox.java — *The environment that will encompass the entire scene*

The files you will modify and submit for this exercise have been `highlighted`. Follow the instructions in the setup documentation to compile and run the source code for this exercise.

You should see a black cube — like the one in Figure 2. You can move the camera by clicking and dragging with your mouse. You can press the `W` key to enable wireframe drawing, and the `P` key to enable vertex-only drawing.

2.1 OpenGL

OpenGL is an application programming interface (API) for the graphics processing unit (GPU). OpenGL contains hundreds of special commands that are used to make a GPU draw shapes. You will use some of the more common

OpenGL commands in this exercise.

The typical OpenGL rendering pipeline is as follows:

1. Specify data for shapes using geometric primitives (generally triangles).
2. Run a *vertex shader* on input primitives to determine their position on the screen, and other optional rendering attributes (e.g. colour).
3. Perform *rasterisation*. This converts geometric primitives into fragments (potential pixels) with locations on the screen.
4. Finally, run a *fragment shader* on each fragment generated by rasterisation to determine its final colour and position on the screen.

For more information, the OpenGL Programming Guide 8th Edition provides a good reference. Make sure to only consult up-to-date documentation.

2.2 LWJGL — Light-Weight Java Games Library

OpenGL implementation is usually provided as a C library and is available on most platforms (Windows, OSX, Linux, Android, iOS). The standard Java libraries do not provide access to OpenGL, so it is necessary to use an external library. In this course, we will use the Lightweight Java Game Library[1] (LWJGL) to provide wrappers around OpenGL's C functions. The Java LWJGL functions work largely the same as the C ones. Note, however, that OpenGL gl* functions are distributed across several packages: org.lwjgl.opengl.GL11, org.lwjgl.opengl.GL15, org.lwjgl.opengl.GL20, org.lwjgl.opengl.GL30. Each package contains functions that were introduced in a given specification of OpenGL, for example GL20 contains all functions introduced OpenGL 2.0. This makes it easier to ensure compatibility with older versions of OpenGL but makes it more difficult to find functions. The easiest solution is to import all packages up to the version we want to support in our application.

The LWJGL library is included in the ZIP file with the template code, in the lib/lwjgl directory.

2.3 JOML — Java OpenGL Math Library

In addition to LWJGL you will also use the JOML library, which contains classes for operations on vectors and matrices. The library is more powerful than the simple Vector class you used in the previous ticks and is in particular intended for OpenGL applications. It is worth checking a few examples at `https://github.com/JOML-CI/JOML/wiki/JOML-and-modern-OpenGL`.

The JOML library is included in the ZIP file with the template code, in the lib/JOML.jar file.

[1] Lightweight Java Game Library 3 – `https://www.lwjgl.org/`

2.4 Compiling and running the code

The instruction below is for compiling and running the code from the command line. Refer to the document *Working with IDEs* for instructions for an IDEs (IntelliJ Idea and Eclipse).

When compiling the code, it is necessary to specify the classpath to the libraries we will use. LWJGL comes in several separate .jar files, some with Java API, other with native libraries. It is recommended that you either use an IDE or create a script for compiling and running the code as command line for both is rather long. To compile, change the current directory to tick2, then run:

```
javac -classpath lib/JOML/joml.jar:lib/lwjgl3.3.1/lwjgl.jar:\
lib/lwjgl3.3.1/lwjgl-glfw.jar:lib/lwjgl3.3.1/lwjgl-opengl.jar \
-d ./out src/uk/ac/cam/cl/gfxintro/crsid/tick2/*.java
```

The -d option specifies where to put the compiled classes. It is a good practice not to mix the sources with compiled code. Note that on Windows you need to replace colons ":" with semicolons ";" in the classpath argument.

Since the program needs to read a few files from the RESOURCES directory, it must be started from the tick2 directory. Moreover, LWJGL library consist of both JAVA classes and a native libraries, which needs to be added to the classpath. You can start the program with the following command:

```
java -classpath lib/JOML/joml.jar:lib/lwjgl3.3.1/lwjgl.jar:\
lib/lwjgl3.3.1/lwjgl-opengl.jar:lib/lwjgl3.3.1/lwjgl-glfw.jar:\
lib/lwjgl3.3.1/lwjgl-natives-linux.jar:\
lib/lwjgl3.3.1/lwjgl-opengl-natives-linux.jar:\
lib/lwjgl3.3.1/l/lwjgl-glfw-natives-linux.jar:./out  \
uk.ac.cam.cl.gfxintro.crsid.tick2.Tick2
```

If you are not runnig on Linux, you need to replace -linux.jar with -windows.jar or -macos.jar. On OSX you may need to add -XstartOnFirstThread argument. Another common issue on OSX is that the application window is hiden behind other windows. If this is the case, use *Mission Control* to find the OpenGL window.

3 What has already been done for you

OpenGL programs can be long and complex, so several steps required for setting up an interactive application have been already implemented for you. This section will walk you through the code so that you know how to extend it. You do not need to modify the code for now. If are already familiar with OpenGL, you can jump directly to Section 4.

3.1 Initializing the application

Let's first look at the OpenGLApplication.initialize() method. The first step is to initialises OpenGL with glfwInit() and create a window with glfwCreateWindow().

```
public void initialize() throws Exception {

if (glfwInit() != true)
  throw new RuntimeException("Unable to initialize the
  ↪  graphics runtime.");

glfwWindowHint(GLFW_RESIZABLE, GLFW_TRUE);

glfwWindowHint(GLFW_SAMPLES, 4); // Multi sample buffer for
↪  MSAA

// Ensure that the right version of OpenGL is used (at least
↪  3.2)
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
glfwWindowHint(GLFW_OPENGL_PROFILE,
↪  GLFW_OPENGL_CORE_PROFILE); // Use CORE OpenGL profile
↪  without deprecated functions
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE); // Make
↪  it forward compatible

window = glfwCreateWindow(WIDTH, HEIGHT, "Tick 2", NULL,
↪  NULL);
if (window == NULL)
  throw new RuntimeException("Failed to create the
  ↪  application window.");
...
}
```

glfwWindowHint() method allows us to set window properties such as ability to be resized, number of samples to be used for anti-aliasing and the version of OpenGL to be used. glfwCreateWindow() finally creates the window of specified dimensions.

The initial position of the window is specified using glfwSetWindowPos(). We then bind the OpenGL context of the window to the current thread using glfwMakeContextCurrent() and define what LWJGL operations are allowed on that context using createCapabilities().

```
{
        ...
        GLFWVidMode mode =
        ↪  glfwGetVideoMode(glfwGetPrimaryMonitor());
        glfwSetWindowPos(window, (mode.width() - WIDTH) / 2,
        ↪  (mode.height() - HEIGHT) / 2);
        glfwMakeContextCurrent(window);
        createCapabilities();
        ...
}
```

The next step is to set OpenGL rendering parameters. In the following code, first v-sync is enabled, rendering of triangles facing away from camera is stopped, multi-sample anti-aliasing is enabled and finally, z-buffer depth testing is enabled.

```
{
        ...
    // Enable v-sync
    glfwSwapInterval(1);

    // Cull back-faces of polygons
    glEnable(GL_CULL_FACE);
    glCullFace(GL_BACK);

    // Enable MSAA
    glEnable(GL_MULTISAMPLE);

    // Do depth comparisons when rendering
    glEnable(GL_DEPTH_TEST);
    ...
}
```

With OpenGL initialised, the next step is to define our scene (camera and meshes). The Camera class allows us to pan around the scene and the CubeRobot class (which you'll be working with) defines the robot's structure and logic. An instance of both classes is created:

```
{
        ...
    // Create camera, and setup input handlers
    camera = new Camera((double) WIDTH / HEIGHT, FOV_Y);
    initializeInputs();

    // This is where we are creating the meshes
    cubeRobot = new CubeRobot();
```

```
    startTime = System.currentTimeMillis();
    currentTime = System.currentTimeMillis();
  }
```

3.2 Initializing the CubeRobot

CubeRobot represents a graphical object that contains the meshes, shaders, textures and their relative transformations required to draw an object that looks like a robot. The following steps take place during CubeRobot initialisation in CubeRobot():

Create the body of this robot represented as a CubeMesh.

```
public CubeRobot() {
  // Create body node

  // Initialise Geometry
  body_mesh = new CubeMesh();
  ...
}
```

CubeMesh contains geometry of a cube generated on the CPU side and stored in four arrays: vertPositions, vertNormals, indices and textureCoordinates. These store the geometry of a cube for now. The four arrays are copied into GPU memory for drawing with OpenGL with loadDataOntoGPU(). For more details on how it is done, refer to Appx. 7.1, Appx. 7.2 and Appx. 7.3.

Next we compile the vertex shader and the fragment shader from their source code files.

```
    {   ...
    // Initialise Shader
    body_shader = new ShaderProgram(new Shader(GL_VERTEX_SHADER,
    ↪  VSHADER_FN), new Shader(GL_FRAGMENT_SHADER, FSHADER_FN),
    ↪  "colour");
    // Tell vertex shader where it can find vertex positions. 3
    ↪  is the dimensionality of vertex position
    // The prefix "oc_" means object coordinates
    body_shader.bindDataToShader("oc_position",
    ↪  body_mesh.vertex_handle, 3);
    // Tell vertex shader where it can find vertex normals. 3 is
    ↪  the dimensionality of vertex normals
    body_shader.bindDataToShader("oc_normal",
    ↪  body_mesh.normal_handle, 3);
    // Tell vertex shader where it can find texture coordinates.
    ↪  2 is the dimensionality of texture coordinates
    body_shader.bindDataToShader("texcoord",
    ↪  body_mesh.tex_handle, 2);
```

```
    ...
    }
```

The shaders are encapsulated into a single ShaderProgram. Shaders tell OpenGL how to render the previously loaded mesh. The body_shader.bindDataToShader() links the mesh data to this shader. For more details on how shaders work, refer to Appx. 7.4.

A PNG image is loaded as texture which can be used later by the shader to colour the mesh. Much of the code has been already written for you. The code for loading, creating, and uploading an OpenGL texture is in the class Texture.

```
{   ...
    // Initialise Texturing
    body_texture = new Texture();
    body_texture.load("resources/cubemap.png");
    ...
  }
```

Finally, a 4x4 transformation matrix is assigned to this body object which instructs shader on how to translate, rotate or scale the object. You'll be writing the code to perform these transformations in your tasks.

```
{   ...
    // Build Transformation Matrix
    body_transform = new Matrix4f();
  }
```

3.3 Rendering with OpenGL

Next, let's look at run(): the main loop, and render(). The `render()` method represents a typical main loop for any interactive graphics application.

```
public void OpenGLApplication.run() {
   initializeOpenGL();
   while (glfwWindowShouldClose(window) != true) {
        render();
   }
}

public void OpenGLApplication.render() {
  // render the scene
  ...
  }
```

The `render()` function is called before every frame and it repeats the following four steps:

Step 1: Clear the buffer

```
{...
  glClearColor(1.0f, 1.0f, 1.0f, 1.0f); // Set the background
  ↪  colour to white
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  ...
}
```

`glClear(...)` clears the OpenGL colour and depth buffers. If we didn't do this, we would re-draw over the previous frame's image.

Step 2: Pass relevant data to the vertex shader.

We first calculate some timings (which will be used later for animation) and then make a call to `CubeRobot.render()`.

```
public void OpenGLApplication.render() {
  ...
    elapsedTime += newTime - currentTime; // Time elapsed since
    ↪  the beginning of this program in millisecs

    float deltaTime = (newTime - currentTime) / 1000.f; // Time
    ↪  taken to render this frame in seconds (= 0 when the
    ↪  application is paused)

    cubeRobot.render(camera, deltaTime, elapsedTime);
  ...
}
```

We need to update the camera position by passing a new model-view-projection matrix to the vertex shader. This step happens in `CubeRobot.renderMesh()`.

```
  public void CubeRobot.render(Camera camera, float deltaTime,
  ↪  long elapsedTime) {
    renderMesh(camera, body_mesh, body_transform, body_shader,
    ↪  body_texture);
  }

  public void CubeRobot.renderMesh(Camera camera, Mesh mesh ,
  ↪  Matrix4f modelMatrix, ShaderProgram shader, Texture
  ↪  texture) {
    // If shaders modified on disk, reload them
    shader.reloadIfNeeded();
    shader.useProgram();

    // Step 2: Pass relevant data to the vertex shader
```

```
    // compute and upload MVP
    Matrix4f mvp_matrix = new
    ↪  Matrix4f(camera.getProjectionMatrix()).mul(camera.getViewMatrix()).mul(modelMatrix);
    shader.uploadMatrix4f(mvp_matrix, "mvp_matrix");

    // Upload Model Matrix and Camera Location to the shader for
    ↪  Phong Illumination
    shader.uploadMatrix4f(modelMatrix, "m_matrix");
    shader.uploadVector3f(camera.getCameraPosition(),
    ↪  "wc_camera_position");

    // Transformation by a nonorthogonal matrix does not preserve
    ↪  angles
    // Thus we need a separate transformation matrix for normals
    Matrix3f normal_matrix = new Matrix3f();
    //TODO: Calculate normal transformation matrix
    shader.uploadMatrix3f(normal_matrix, "normal_matrix");
  ...
}
```

The code above calculates the model view projection matrices and passes them to the vertex and fragment shader. Refer to comments in the code for details on each operation.

Step 3: Draw our VertexArray as triangles.

```
public void CubeRobot.renderMesh(...) {
  ...
    // Bind Texture
    texture.bindTexture();
    // draw
    glBindVertexArray(mesh.vertexArrayObj); // Bind the existing
    ↪  VertexArray object
    glDrawElements(GL_TRIANGLES, mesh.no_of_triangles,
    ↪  GL_UNSIGNED_INT, 0); // Draw it as triangles
    glBindVertexArray(0);              // Remove the binding

    // Unbind texture
    texture.unBindTexture();
  }
```

`glDrawElements()` draws the geometry associated with the currently bound `VertexArray`, which we loaded earlier with `loadDataOntoGPU()`. Before calling it, relevant texture is binded to the GPU using `texture.bindTexture()`. The mesh is rendered to the back buffer.

Step 4: Swap buffers. Since our window is double-buffered, we swap the front and back buffers to display the fully-drawn image to the user. Then we check for

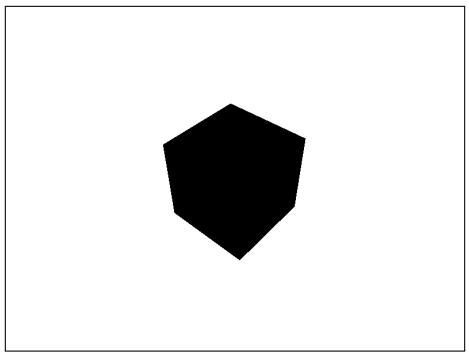


Figure 3: Simple black cube without any illumination.

any input events, e.g. mouse movements, with `glfwPollEvents()` and finally, we check if any OpenGL errors have occurred.

```
public void OpenGLApplication.render() {
  ...
  glfwSwapBuffers(window);
  glfwPollEvents();
  checkError();
}
```

4 Exercises

The skeleton code renders a simple black cube without any illumination (Figure 3). Using this not-so-exciting animation as a starting point, you will add Phong's illumination model (Sec. 4.1), and additional cubes to the scene so that they form a robot (Sec. 4.2). Then you will animate the robot (Sec. 4.3 and 4.4), put a texture on it (Sec. 4.5), and finally add an environment map to simulate perfect specular reflections on the robot (Sec. 4.6).

4.1 Shading

Files to be edited: `CubeRobot.java`, `cube_vertex_shader.glsl`, `cube_fragment_shader.glsl`

In this exercise, you will add lighting to your scene and implement the Phong illumination model just like you did in Tick 1. For this, you will write two shader programs.

A shader is a short program that is executed on the GPU. They are written in GLSL – the OpenGL Shading Language, which is similar to both C and Java. The complete specification of the GLSL language can be found at `https:`

//www.opengl.org/documentation/glsl/. However, for this exercise you only need to know a small set of GLSL instructions, explained in the lectures and below. To draw any shape in OpenGL, we need two shaders: a vertex shader, which processes vertices, and a fragment shader, which processes fragments (pixels).

The main job of the vertex shader is to transform and project the 3D world coordinates of each vertex into 2D screen coordinates. The 2D coordinates and depth must be stored in the special variable gl_Position. gl_Position is an output variable, which does not need to be explicitly declared, but must be always assigned by the vertex shader.

Code Structure:

Let us take a look at the starting code for the vertex shader in `cube_vertex_shader.glsl`:

```
#version 140

in vec3 oc_position;          // vertex position in object
↪  coordinates (oc_)
in vec3 oc_normal;           // vertex normal in object
↪  coordinates
in vec2 texcoord;       // texture coordinate in texture space

out vec3 wc_frag_normal;     // fragment normal in world
↪  coordinates
out vec2 frag_texcoord;       // fragment texture coordinate in
↪  world coordinates
out vec3 wc_frag_pos;      // fragment position in world
↪  coordinates

uniform mat4 mvp_matrix;    // model-view-projection matrix
uniform mat4 m_matrix;     // model matrix
uniform mat3 normal_matrix;  // inverse transpose of the model
↪  matrix

void main()
{
    frag_texcoord = texcoord;  // We just copy texture
    ↪  coordinates and pass to fragment shader

    // TODO: Transform vertex from local to world coordinates
    // TODO: Transform vertex normal to world coordinates
```

```
    // The position is projected to the screen coordinates using
    ↪  mvp_matrix
    gl_Position = mvp_matrix * vec4(oc_position, 1.0);
}
```

The 3D coordinates of each vertex, read from `oc_position` input variable, are transformed by the pre-computed 4×4 model-view-projection matrix `mvp_matrix`. Note that the `oc_position` needs to be transformed from the conventional Cartesian to homogeneous coordinates before it can be transformed. For that, you will need to create a vector of type `vec4`.

`mvp_matrix` is a uniform variable. This variable remains the same for each processed vertex (thus uniform), but it can be changed from Java code before a draw instruction is enqueued. Let's take a look at the code for changing uniforms, which you can find in CubeRobot.renderMesh():

```
public void renderMesh(Camera camera, Mesh mesh , Matrix4f
↪  modelMatrix, ShaderProgram shader, Texture texture) {
  ...
  // Step 2: Pass relevant data to the vertex shader

  // compute and upload MVP
  Matrix4f mvp_matrix = new
  ↪  Matrix4f(camera.getProjectionMatrix()).mul(camera.getViewMatrix()).mul(modelMatrix);
  shader.uploadMatrix4f(mvp_matrix, "mvp_matrix");

  // Upload Model Matrix and Camera Location to the shader for
  ↪  Phong Illumination
  shader.uploadMatrix4f(modelMatrix, "m_matrix");
  shader.uploadVector3f(camera.getCameraPosition(),
  ↪  "wc_camera_position");

  // Transformation by a nonorthogonal matrix does not preserve
  ↪  angles
  // Thus we need a separate transformation matrix for normals
  Matrix3f normal_matrix = new Matrix3f();
  //TODO: Calculate normal transformation matrix
  shader.uploadMatrix3f(normal_matrix, "normal_matrix");

  ...
}
```

The model-view-projection matrix is computed in Java code as the multiplication of the projection matrix by the view matrix (the model matrix is an identity matrix in our example). Note that the transformation that is performed first on the vertex (view) must be right-most in the matrix product. You may be surprised to see that we are computing a matrix multiplication in Java (on the CPU) when the GPU can perform this operation much faster. The reason is

that we compute the uniform variable only once for the entire object, while in the vertex shader this operation would be repeated for each vertex.

For each fragment, you will need to know its position and normal vector in world coordinates in order to calculate colour according to Phong's illumination model. As discussed in the lectures, normal vectors need to be transformed differently than vertices as transformation by a non-orthogonal matrix does not preserve angles.

The fragment shader's job is to set the `color` variable for each fragment — the colour of the pixel that will be drawn on the screen. This is where you will implement shading.

Let us look at the starting source code of the fragment shader in cube_fragment_shader.glsl:

```
#version 140

in vec3 wc_frag_normal;             // fragment normal in world
↪  coordinates (wc_)
in vec2 frag_texcoord;      // texture UV coordinates
in vec3 wc_frag_pos;      // fragment position in world
↪  coordinates

out vec3 color;         // pixel colour

uniform sampler2D tex;      // 2D texture sampler
uniform vec3 wc_camera_position;  // Position of the camera in
↪  world coordinates

// Tone mapping and display encoding combined
vec3 tonemap(vec3 linearRGB)
{
    float L_white = 0.7; // Controls the brightness of the image

    float inverseGamma = 1./2.2;
    return pow(linearRGB/L_white, vec3(inverseGamma)); // Display
    ↪  encoding - a gamma
}

void main()
{
  vec3 linear_color = vec3(0, 0, 0);
  // TODO: Calculate colour using Phong illumination model
  // linear_color = ambient + diffuse + specular
  ...

  color = tonemap(linear_color);
}
```

First thing to note is that the names of the input variables in the fragment shader (`wc_frag_normal`, `wc_frag_pos`, and `frag_textcoord`) are the same as the names of the output variables in the vertex shader. This is how OpenGL knows how to link input/output from both shaders. Computing colour is split into two steps: first we compute the linear colour value, then we apply a tone-mapping function to transform linear values into a gamma-encoded pixel values. Tone mapping and the rationale for this step will be explained in the last lecture.

Tasks:

- In the above function (`renderMesh()` in the file `CubeRobot.java`), compute a 3x3 normal transformation matrix from model matrix and replace the current normal matrix with that.
- In the vertex shader (`resources/cube_vertex_shader.glsl`), use this normal matrix to calculate the fragment normals `wc_frag_normal`. Also, calculate the fragment position in world space `wc_frag_pos` using the model matrix. This will be used for shading.
- Calculate the RGB vector `color` using the Phong illumination model. Start by adding a point light source in your fragment shader `resources/cube_fragment_shader.glsl`. You can do this by creating a constant with a position of the light source. Use it along with the fragment normal and fragment position you computed in the vertex shader to calculate reflected ray, viewing direction and incident direction.
  Define variables with the ambient term, diffuse constant and specular constant. Finally, calculate the fragment colour as the sum of ambient light, diffuse light and specular light. You can refer to Tick 1 description on how to do this.

Hint: Look up the documentation of the following GLSL functions: `normalize()`, `reflect()`, `max()`, `dot()` and `pow()`. You may also want to check the matrix operations supported by JOML: `https://joml-ci.github.io/JOML/apidocs/org/joml/Matrix4f.html`.

You can use the values provided below for the shading variables to recreate Figure 4. This will help you test your implementation but we encourage you to experiment with different values.
Light position = $(-1, 3, -1)$
Light color = $(0.941, 0.968, 1)$
ambient light $I_a = (0.16, 0, 0)$
diffuse constant $k_d = 0.4$
specular constant $k_s = 0.75$
Diffuse color $C_{diff} = (1, 0, 0)$
Specular color $C_{spec} = (1, 1, 1)$
Phong's roughness coefficient $\alpha = 32$

Note: You do not need to compile your program again when you make changes to your shader. If you change the shader code while the application is running, the application will automatically reload and recompile the new shader (unless the shader fails to compile).

Note:* If you face any trouble in this exercise, you may refer to Sec 7.5 to learn how to debug shaders.

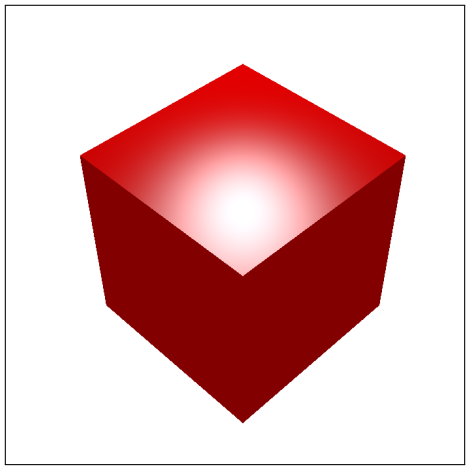


Figure 4: Phong Shading.

4.2 Transformations

Files to be edited: `CubeRobot.java`

A cube is not really an exciting shape to look at. Let's turn this cube into a more humanoid looking robot. You will work in `CubeRobot.java` file.

Code Structure

The relevant part of the code in CubeRobot.java is:

```
...

// TODO: Add Component 2: Right Arm

...

public CubeRobot() {
  // Create body node
```

```
  ...
  // Initialise Texturing
  body_texture = new Texture();
  body_texture.load("resources/cubemap.png");

  // Build Transformation Matrix
      body_transform  = new Matrix4f();

      // TODO: Scale the body transformation matrix

  // TODO: Create right arm node

  ...

  // TODO: Build Transformation Matrix (rotate the right arm
  ↪  around its end)

}

public void CubeRobot.render(Camera camera, float deltaTime,
↪  long elapsedTime) {

  ...

  renderMesh(camera, body_mesh, body_transform, body_shader,
  ↪  body_texture);

  // TODO: Render Arm.

...

}
```

Transformation matrices are useful for moving, rotating and scaling meshes. The body cube's transformation matrix (`body_transform`) serve as the basis of the robot's body. This matrix, also known as the model matrix, is uploaded to the vertex shader (along with view and projection matrix), where it is used to calculate the vertex and fragment position in world space. *Ignore for now the code related to textures. You will add textures in a later task.*

Tasks:

- Scale the cube into an elongated cuboid by updating the body cube's transformation matrix such that it is elongated along the y-direction (Figure 5).

You can use matrix operations provided by JOML library.

- Create another cube for the robot's right arm and render it (refer to how the body was created and rendered). Scale it such that its proportions matches those of the body. Then, translate it so that it is at the correct position on the body, and rotate it around its edge as shown in (Figure 6).

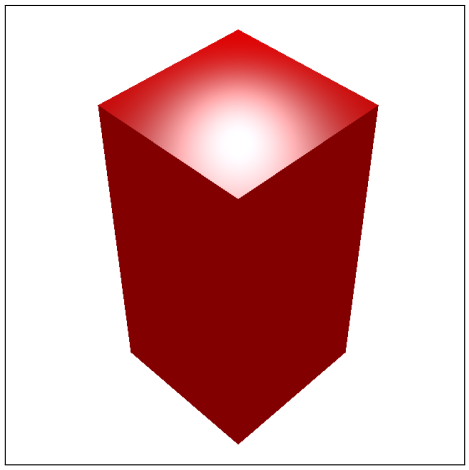


Figure 5: Scaled cube

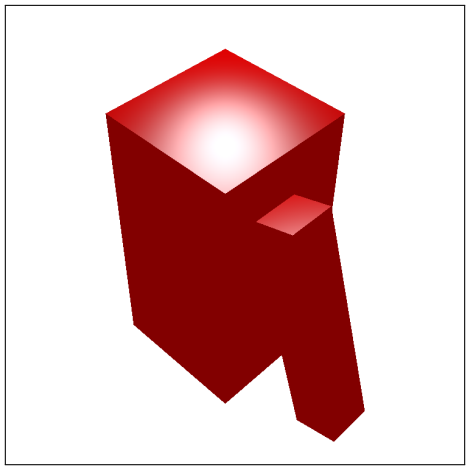


Figure 6: Add an arm to the body

You have to be careful while ordering your transformation operations as they are not commutative.

Hint: Look at the `translate()`, `rotateAffineXYZ()`, and `scale()` member functions of `Matrix4f` class on JOML's online documentation.

You can use the values given below to recreate Figure 5 and Figure 6:
Body scale $(S_x, S_y, S_z) = (.75, 1.5, 0.75)$
Arm scale $(S_x, S_y, S_z) = (.25, 1.25, .25)$
Arm rotation $(\theta_x, \theta_y, \theta_z) = (0, 0, 0.3)$[radian] about the hinge at $(0, 1, 0)$
Arm translation $(x, y, z) = (1, -0.25, 0.125)$

4.3 Animation

Files to be edited: `CubeRobot.java`

The robot is starting to come in shape. Now, let us make it dance!

The transformation matrices for the body and the arm are passed to the shaders in an infinite loop. What will happen if we update those matrices in every iteration? You get animation!

Task:

- Rotate the body around the y-axis and the arm around its hinge as a function of time. Use the arguments `deltaTime` (difference in time between two frames) or `elapsedTime` (the absolute time) as the parameter controlling the progress of your animation. Note that an animation can be rendered at different refresh-rates and it is important to control the speed of the animation according to the physical time rather than the frame number.

```
/**
 * Updates the scene and then renders the CubeRobot
 * @param camera - Camera to be used for rendering
 * @param deltaTime    - Time taken to render this frame in
↪  seconds (= 0 when the application is paused)
 * @param elapsedTime  - Time elapsed since the beginning of
↪  this program in millisecs
 */
public void CubeRobot.render(Camera camera, float deltaTime,
↪  long elapsedTime) {

  // TODO: Animate Body. Translate the body as a function of
  ↪  time

  // TODO: Animate Arm. Rotate the left arm around its end as a
  ↪  function of time

  renderMesh(camera, body_mesh, body_transform, body_shader,
  ↪  body_texture);
```

```
    ...

}
```

We obviously want the arm to be firmly attached to the body and move along with it when the body is animated. The arm would look very unnatural if it dangled independently (Figure 7). Check the supplementary video `Task_4.3.mp4` to see a sample of what you should expect. The best way to ensure that the individual components of your animated character move consistently is to organize them into a scene graph.

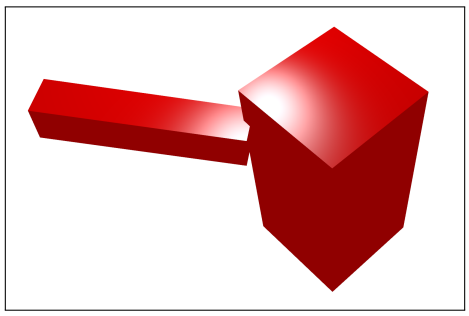


Figure 7: Body and arm rotating independently

4.4 Scene Graph

Files to be edited: `CubeRobot.java`

You only have two objects in the scene, but when you start adding many more objects, keeping track of the relative transformations becomes very difficult. Scene graphs are a way to group and chain the transformation matrices of all the objects in a scene so that they form a hierarchy: when parent objects are animated, the child objects move together with their parents. This is achieved by multiplying the (model) transformation matrices of the parent and child, so that the resulting transformation combines both. By doing this in a chain, you can have a very complex scene with many objects moving together without having to individually update the position of each object in the scene.

```
public void CubeRobot.render(Camera camera, float deltaTime,
↪  long elapsedTime) {

  ...
  renderMesh(camera, body_mesh, body_transform, body_shader,
  ↪  body_texture);
  // TODO: Chain transformation matrices of the arm and body
  ↪  (Scene Graph)
```

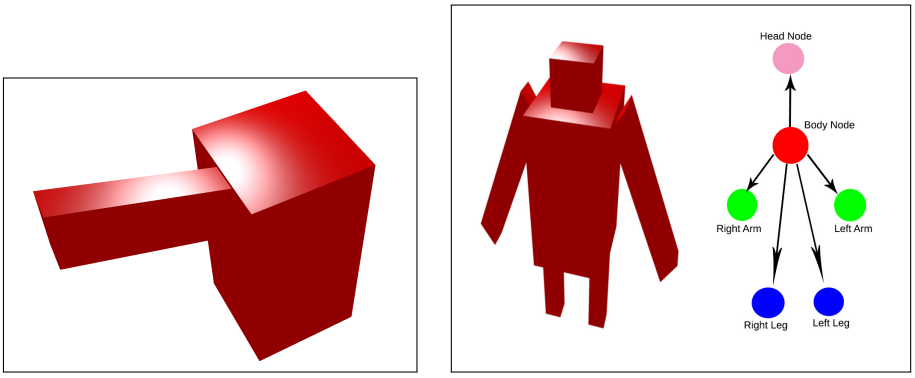
```
    // TODO: Render Arm.
    ...

}
```

Tasks:

- Your first task is to update the transformation matrix of your arm object such that it stays attached to the body when the body rotates (Figure 8a).
- You now have all the necessary knowledge to complete the robot. Add more parts to your robot's body and link their transformation matrices as shown in Figure 8b. You are encouraged to use a data structure such as a graph or a stack to store the relationships between the robot parts. However, you can also explicitly define the transformations in the code.

Please refer to supplementary videos `Task_4.4.1.mp4` and `Task_4.4.2.mp4` for the expected output of this task.



(a) Body and arm rotating together (b) Scene graph of the robot

Figure 8: Animation using Scene Graph

4.5 Texturing

Files to be edited: `cube_fragment_shader.glsl`

You can set the colour of the cube by manipulating the output colour in the fragment shader. You can paint it with a single colour or procedurally generate patterns on it. However, that takes a considerable amount of effort and limits your artistic freedom. A solution is to use a texture.
A texture is an image (for 2D textures) that can be used to add details to an object. Think of it as a 2D sheet of paper that can be wrapped on a 3D mesh.

Code Structure:

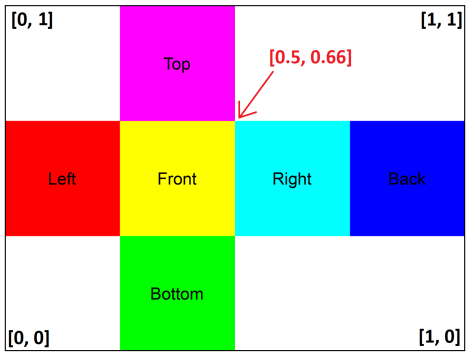


Figure 9: UV mapping of the cube.

In order to map a texture to a mesh, we need to tell each vertex of the mesh which part of the texture (specified by texture coordinate) it corresponds to. When rasterizing a triangle, the texture coordinates will be interpolated between vertices (using barycentric coordinates) and passed to the fragment shader.

Texture coordinates range from 0 to 1 in the x and y-axis (remember that we use 2D texture images). Retrieving the texture colour using texture coordinates is called sampling. Texture coordinates start at (0,0) for the lower-left corner of a texture image to (1,1) for the upper-right corner of a texture image. Figure 9 shows how we map texture coordinates to our cube. Exact mapping of this texture can be found in the function `CubeMesh.initializeTextureCoordinates()`:

```
float[] initializeTextureCoordinates() {
  float[] texCoors = new float[] {
      1/4f,0,   1/4f,1/3f,   1/2f,0,
      1/4f,1/3f,  1/2f,1/3f,  1/2f,0,
      1/4f,2/3f,  1/4f,1,    1/2f,2/3f,
      1/4f,1,    1/2f,1,    1/2f,2/3f,
      1/4f,1/3f,   1/4f,2/3f,  1/2f,1/3f,
      1/4f,2/3f,  1/2f,2/3f,  1/2f,1/3f,
      1,2/3f,    1,1/3f,   3/4f,2/3f,
      1,1/3f,    3/4f,1/3f,  3/4f,2/3f,
      0,2/3f,    1/4f,2/3f,  0f,1/3f,
      1/4f,2/3f,  1/4f,1/3f,  0f,1/3f,
      3/4f,1/3f,  1/2f,1/3f,  3/4f,2/3f,
      1/2f,1/3f,  1/2f,2/3f,  3/4f,2/3f
  };
  return texCoors;
}
```

For convenience, we store a texture as a PNG image, which can be easily edited. It needs to be uncompressed and rearranged so that it is in a format expected by OpenGL (row-major, interleaved colour channels). This is done in the load() function of the Texture class. It first reads the image file and converts it into a byte buffer. It then asks OpenGL to create a texture memory using glGenTextures() function and binds it using glBindTexture() so that all subsequent texture commands work on the currently bound texture. We then upload our previously loaded image data to this texture using glTexImage2D. This is followed by specifying the mipmapping and scaling strategies to be used on this texture. All those steps are done in the `Texture.load()` method shown below.

```
public int load(String filename) {
  ...

  // Link the PNG decoder to this stream
  BufferedImage image = loadImageFromFile(filename);

  ...

  // Decode the PNG file in a ByteBuffer
  buffer = ByteBuffer.allocateDirect(4 * image.getWidth() *
  ↪  image.getHeight());
  ...

  // Create a new texture object in memory and bind it
  texId = GL11.glGenTextures();
  GL13.glActiveTexture(GL13.GL_TEXTURE0);
  GL11.glBindTexture(GL11.GL_TEXTURE_2D, texId);

  // All RGB bytes are aligned to each other and each component
  ↪  is 1 byte
  GL11.glPixelStorei(GL11.GL_UNPACK_ALIGNMENT, 1);

  // Upload the texture data.
  // Note that the internal texture format is sRGB since the
  ↪  shading is in the linear colour domain.
  GL11.glTexImage2D(GL11.GL_TEXTURE_2D, 0, GL21.GL_SRGB8, tWidth,
  ↪  tHeight, 0, GL11.GL_RGBA, GL11.GL_UNSIGNED_BYTE, buffer);


  // Setup what to do when the texture has to be scaled
  GL11.glTexParameteri(GL11.GL_TEXTURE_2D,
  ↪  GL11.GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
  GL11.glTexParameteri(GL11.GL_TEXTURE_2D,
  ↪  GL11.GL_TEXTURE_MAG_FILTER, GL11.GL_LINEAR);
```

```
  GL11.glTexParameteri(GL11.GL_TEXTURE_2D,
  ↪  GL11.GL_TEXTURE_WRAP_S, GL12.GL_CLAMP_TO_EDGE);
  GL11.glTexParameteri(GL11.GL_TEXTURE_2D,
  ↪  GL11.GL_TEXTURE_WRAP_T, GL12.GL_CLAMP_TO_EDGE);

  GL30.glGenerateMipmap(GL11.GL_TEXTURE_2D);

  GL11.glBindTexture(GL11.GL_TEXTURE_2D, 0); // Unbind texture
  ↪  from the current context

  return texId;
}
```

Note that the texture internal format is set to `GL21.GL_SRGB8`. Such textures are stored in gamma-corrected (gamma-encoded) pixel format but are converted to a linear colour space when sampled. You will learn more about gamma-corrected and linear colour spaces in the lecture on colour.

Since an application could use multiple textures, we need to make sure we have bound the correct texture while rendering a mesh. This is done using `texture.bindTexture()` in `CubeRobot.renderMesh()` function. **Note:** You can bind multiple textures at the same time but the upper bound for this is limited by your GPU. In this exercise, you will only work with one texture per mesh. Also, GPU memory is costly and limited. If two meshes share the same texture image, try to only load it onto memory once and use the same texture for both.

We need to provide the fragment shader the access to this texture. GLSL has a built-in data-type for texture objects called a sampler. The various sampler types are separated into 3 categories: sampler1D, sampler2D and sampler3D. Since we are working with 2D images, we will use sampler2D. We can then add a texture to the fragment shader by simply declaring a uniform sampler2D. As this is the only texture you are using in this shader, you do not need to explicitly assign this sampler to the texture that you bind in `CubeRobot.renderMesh()`.

```
...
// cube_fragment_shader.glsl
uniform sampler2D tex;        // 2D texture sampler
...
void main()
{

  vec3 linear_colour = vec3(0,0,0);
  // TODO: Calculate colour using phong illumination model
  // colour = ambient + diffuse + specular
  // TODO: Sample the texture and replace diffuse surface colour
  ↪  (C_diff) with texel value
```

```
    ...

    colour = tonemap( linear_colour );
}
```

Tasks:

- Sample the 2D texture `tex` to find the the diffuse colour of the surface (C_{diff}) in your Phong illumination calculation. To sample the colour of a texture, look up the documentation of GLSL's built-in `texture()` function. You will make changes in `resources/cube_fragment_shader.glsl`.
- Use an image editor of your choice to update the cubemap texture and add some interesting details to the robot. As an example, we have provided you a texture `resources/cubemap_head.png` for your robot's head. Load and bind it to your head's cube mesh during CubeRobot initialization (`CubeRobot()`).

If you did it correctly, your result will look like in Figure 10.

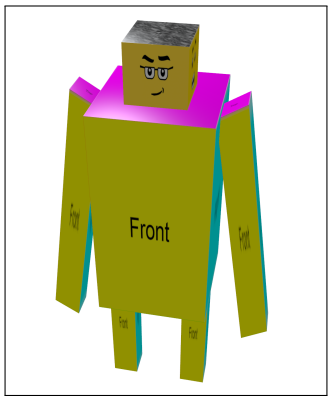


Figure 10: A textured cube.

Try different filtering options in `Texture.load()` and see how the output changes.

4.6 Skybox and reflections

Files to be edited: `cube_fragment_shader.glsl`

A dancing robot is good; a dancing robot in the streets of Cambridge is better. In this exercise you will learn about a new type of texture called *cubemaps* and how they can be used to make your scene much more immersive by creating skyboxes and reflections.

A cubemap is a texture made up of 6 separate 2D textures, each corresponding to a different side of a cube. It has a unique property, it is sampled using direction vector. When you sample a cubemap using a direction vector, OpenGL assumes a hypothetical 1x1x1 cube with the 6 sides made up of different 2D textures and origin of the direction vector to be the center of that cube (Figure 11). The magnitude of the direction vector does not matter. This means that if we have a cube mesh centered at the origin, vertices on this mesh are the same as direction vectors used to sample the cubemap texture. This property can be used to determine the texture coordinates in the vertex shader as demonstrated later in the code structure.

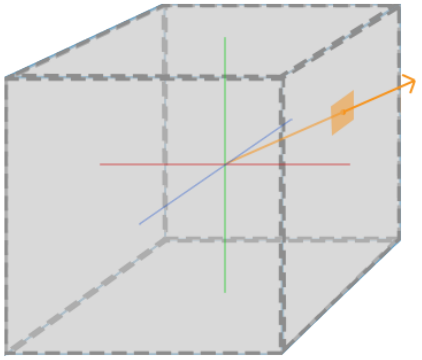


Figure 11: Sampling a cubemap. *Source https://learnopengl.com/Advanced-OpenGL/Cubemaps*

If we put a cubemap on a very large cube that encompasses our current scene, we can create an illusion of our robot being in a much larger environment than it actually is. This technique is called **skybox**. Use of skyboxes is very common in video games and CGI movies where they are used to simulate hard to create environments. In this exercise, you will use a skybox that uses photos of Cambridge streets as a cubemap.

Having a cubemap in your scene also enables another very useful effect: perfect specular reflections.

Since we know the viewing direction and the normal at a given fragment position, we can determine the direction of the reflected ray and use the cubemap to sample the colour of the incoming light (see Figure 12). This is a one-bounce approximation of the recursive ray-tracing we used in Tick 1. Note that this kind of reflection will ignore reflections from any objects in the scene and will take colour directly from the cubemap. The reflected colour is combined with the other reflection components contributing to the surface (diffuse and specular).

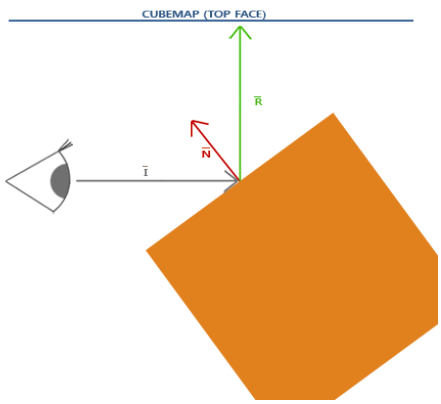


Figure 12: Reflections using cubemap.*Source https://learnopengl.com/Advanced-OpenGL/Cubemaps*

Code Structure:

Most of the code in this exercise is abstracted for you in the `Skybox.java` class. Let's walk-through the code.

```
public class SkyBox {

    ...
  public SkyBox() {
    // Create body node

    // Initialise Geometry
    skybox_mesh = new CubeMesh();

    ...

    // Initialise Texturing
    skybox_texture = new Texture();
    String filenames[] = {"resources/skybox/right.png",
        "resources/skybox/left.png",
        "resources/skybox/top.png",
        "resources/skybox/bottom.png",
        "resources/skybox/front.png",
        "resources/skybox/back.png"
        };
    skybox_texture.loadCubemap(filenames);
  }
  ...
```

```
}
```

Skybox class is very similar to CubeRobot class. The key difference is that it is made up of a single cube and uses cubemap texture instead of 2D textures. For more details on how cubemaps are loaded, refer to `Texture.java`.

Unlike CubeRobot, we want to draw the insides of our cube mesh. To prevent OpenGL from automatically culling the back of a triangle, we need to disable culling before the render call. Also, since our skybox will always be behind the actual scene, we can prevent writing to the z-buffer by calling `GL11.glDepthMask(false)`.

```
    ...
  public void SkyBox.render(Camera camera, float deltaTime, long
  ↪  elapsedTime) {

    // Disable culling of polygon back-faces to draw the inside
    ↪  of the cube
    GL11.glDisable(GL_CULL_FACE);
    // Disable depth writing so that the skybox is always be
    ↪  drawn at the background
    GL11.glDepthMask(false);
    // Skybox cube needs to be scaled a size where it contains
    ↪  the entire scene
    renderMesh(camera, skybox_mesh, new Matrix4f().scale(new
    ↪  Vector3f(10f,10f,10f)), skybox_shader, skybox_texture);
    GL11.glDepthMask(true);
    glEnable(GL_CULL_FACE);
    glCullFace(GL_BACK);
  }
```

As we discussed before, the position on the cube can be used as direction vector for the sampling the cube map texture. We set the texture coordinates to be the same as position coordinates in the vertex shader which are then sampled in the fragment shader. Since we do not want the skybox to move with the camera and only rotate, we remove the translation component of the camera view matrix.

```
...
public void SkyBox.renderMesh(Camera camera, Mesh mesh , Matrix4f
↪  modelMatrix, ShaderProgram shader, Texture texture) {
  ...

  // compute and upload MVP
  // Skybox should not move with camera, only rotate
  Matrix4f view_matrix = camera.getViewMatrix();
  view_matrix.setTranslation(new Vector3f(0,0,0));
```

```
  Matrix4f mvp_matrix = new
  ↪  Matrix4f(camera.getProjectionMatrix()).mul(view_matrix).mul(modelMatrix);;
  shader.uploadMatrix4f(mvp_matrix, "mvp_matrix");

  ...
}

...
 // skybox_vertex_shader.glsl
   ...

   void main()
   {
       frag_texcoord = oc_position;
       // The position is projected to the screen coordinates
       ↪  using mvp_matrix
       gl_Position = mvp_matrix * vec4(oc_position, 1.0);
   }

// skybox_fragment_shader.glsl
...

uniform samplerCube skybox;       // Cubemap texture sampler
...
void main()
{
    // TODO: Sample the skybox to determine the colour of the
    ↪  environment
    vec3 linear_color = vec3(1,1,1);
  color = tonemap(linear_color);
}
```

Tasks:

- Skybox: Your first task is similar to previous exercise. Sample the cubemap texture in `skybox_fragment_shader.glsl` to determine the colour of the environment.
- Reflections: The same cubemap is also available in `cube_fragment_shader.glsl`. Calculate the reflection direction of each fragment using the view and fragment position and sample the cubemap using this direction. Multiply this colour with a fraction and add it to your previously calculated robot colour,

If you did it correctly, your results should look like Figure 13.

Tip: Press *Space* key to pause your animation and move the camera using your

mouse. This will help you debug the reflections.

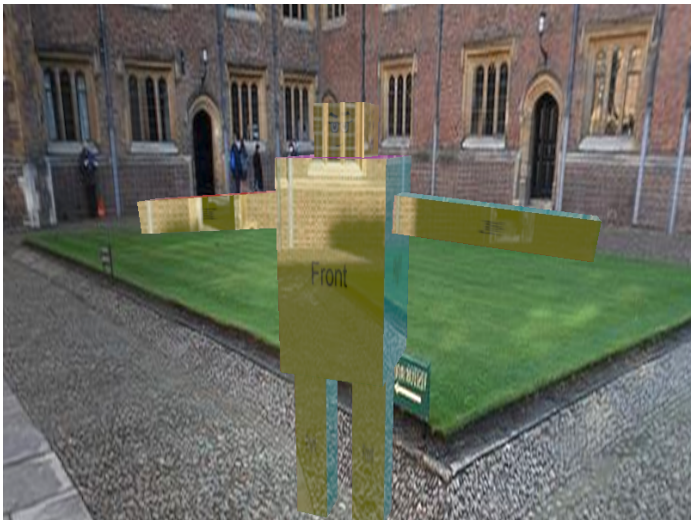


Figure 13: Adding a skybox and reflections to the scene

5 Tick2*: Fresnel Equations (optional)

Before attempting this exercise, make sure you have completed all the previous ones.

In the last step of the regular tick, you gave mirror like attributes to your robot by adding reflections. In a very similar fashion you can also make appear 'glass-like' by simulating refraction. The direction of the refracted ray is calculated using the Snell's law. GLSL provides a helper function `refract()` that takes care of this for you. In Tick2* you will make the robot stand on a large plate of glass.

In real life, most objects are neither fully-reflective nor fully refractive. Imagine standing next to a lake, you will be able to see the lake bed through the water near you, while the water further away will appear to reflect the surroundings. This ratio of the amount reflected vs. refracted light can be approximated using Fresnel equations. If you want to learn more about the effect they model, check Chapter 13 in the Fundamentals of Computer Graphics. The two fresnel equations are:

$$R_s = \left| \frac{\eta_1 \cos\theta_i - \eta_2 \cos\theta_t}{\eta_1 \cos\theta_i + \eta_2 \cos\theta_t} \right|^2 \tag{1}$$

$$R_p = \left| \frac{\eta_1 \cos\theta_t - \eta_2 \cos\theta_i}{\eta_1 \cos\theta_t + \eta_2 \cos\theta_i} \right|^2 \tag{2}$$

where, R_s and R_p are the reflectance of s-polarized and p-polarized light, η_1 and η_2 are the refractive indices of the two media, θ_i is the angle of incidence/reflection and θ_t is the angle of transmission/refraction. The fraction of reflected light is given by the average of these equations.

$$F_R = \frac{1}{2}\left(R_s + R_p\right) \tag{3}$$

Since the amount of total energy is conserved, the fraction of refracted/transmitted light is

$$F_T = 1 - F_R \tag{4}$$

Task

- Add a cube below your robot and scale it such that it looks like a large floor beneath your dancing robot. This floor will provide a good demonstration of fresnel reflection/transmission (similar to the lake example).
- Calculate the colour of reflected and refracted rays for each fragment in `cube_fragment_shader.glsl`. Solve the fresnel equations to calculate the ratio of reflection vs refraction and update the floor's colour accordingly.

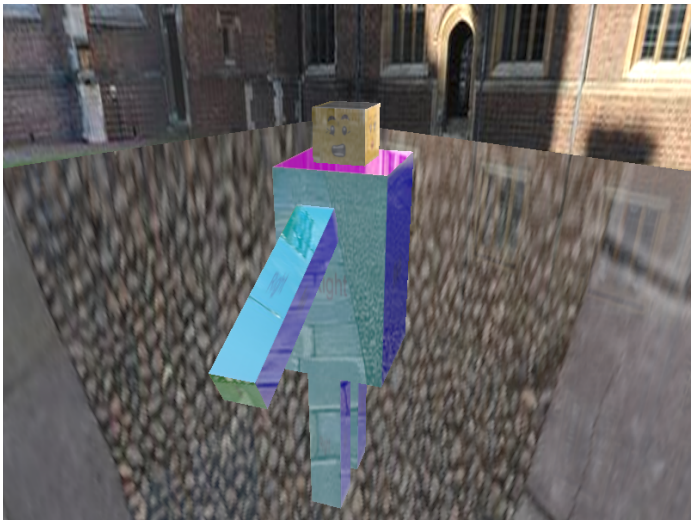


Figure 14: Fresnel Reflections/Transmission

Your results should look like Figure 14. Notice that the floor is mostly refractive when looking directly down but gets more reflective as you move towards the edges.

Submit a screenshot and all Java files that you have modified in a single ZIP file.

6 Submission

Please generate the output image `screenshot.png` with an initial frame of your animation. To generate this image, execute the application with "-o screenshot.png" argument.

You need to submit following directories and file for this tick:

- `src/`
- `resources/`
- `screenshot.png`

Please put both the directories and the file in a ZIP archive (no RAR or 7z) and submit as a single file. There is no automated testing for this task. We will be compiling and testing the submissions and/or checking them in a ticking session. Note that you if you are selected for ticking, you will need to demonstrate the running application.

7 Appendix

The appendix gives you further details on how to work with OpenGL code. The information contained here is not necessary to complete the Tick, but could be useful if you are interested in OpenGL programming and you would like to further expand the code.

7.1 Preparing vertex data

In order to draw any primitives, OpenGL must be provided with the geometry: vertex attributes and indices of the vertices that form primitives (usually triangles).

OpenGL stores geometry in several Buffers, which are illustrated as an UML class diagram in Figure 15. Each individually drawn object, or its part, is represented by a VertexArray. The VertexArray is a container for several buffers, which could be either ArrayBuffers or ElementArrayBuffers. ArrayBuffer stores vertex attributes, such as positions, normals, colours, texture coordinates, or any user-defined data. ElementArrayBuffer stores integer indices into ArrayBuffers. Those indices define the vertices of primitives to be drawn, for example three indices of the triangle vertices.

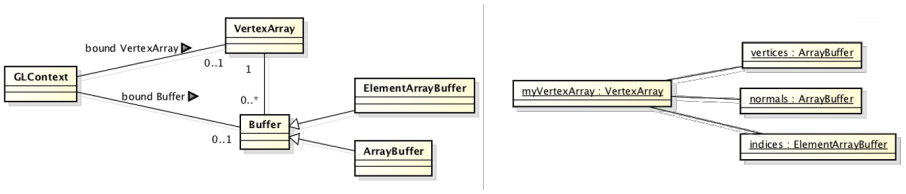
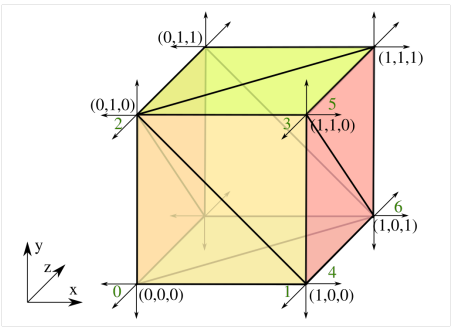


Figure 15: OpenGL objects used to store geometry. The diagram uses the UML notation. See text for details.



cube::ElementArrayBuffer
0, 1, 2, 2, 1, 3, 4, 6, 5, ...

index	pos::ArrayBuffer	normals::ArrayBuffer
0	0, 0, 0,	0, 0, -1
1	1, 0, 0,	0, 0, -1
2	0, 1, 0,	0, 0, -1
3	1, 1, 0,	0, 0, -1
4	1, 0, 0,	1, 0, 0
5	1, 1, 0,	1, 0, 0
6	1, 0, 1,	1, 0, 0

Figure 16: A cube represented as a triangle mesh. Each face is split into two triangles. Normals (short arrows) need to be specified separately for each face of the cube and therefore vertex attributes cannot be shared between triangles that belong to different faces. The green numbers represent indices of the vertex attributes.

For example, if we were to represent the cube shown in Figure 16 as a triangle mesh, we would need to prepare two ArrayBuffers, one with vertex positions, the other with vertex normals, and one ElementArrayBuffer with the indices of the vertex attributes. Indices allow vertex attributes to be shared between triangles. For example, two triangles on the front face can share vertices labeled as 1 and 2 (green numbers in Figure 16). However, vertex attributes cannot be shared between triangles laying on different faces. This is because those triangles have different normals. This is the reason why some vertex positions need to be duplicated in the ArrayBuffer.

The tables on the right in Figure 16 show how the first three triangles of the cube can be specified in OpenGL arrays. The indices of vertex attributes must be specified in the counter-clockwise order. OpenGL uses that order to determine whether a triangle is facing the camera and is visible, or is facing away from the camera and can be skipped from drawing.

7.2 Managing buffers in OpenGL

The core API of OpenGL was designed in the 90s without much regard to Object Oriented design patterns. For that reason, the way in which OpenGL manages different buffers and resources may confuse even experienced programmers. However, it is enough to learn a few patterns to get comfortable with most OpenGL oddities.

In order to create an OpenGL object, we first need to generate a "name" using one of the glGen* functions. A "name" can be understood as a reference or pointer to the object, which is represented as a integer number. glGen* calls do not allocate an object itself or any memory associated with it. To allocate the actual object, we need to call one of the glBind* functions. glBind* functions perform two operations: they allocate a new object, but only if it has not been allocated before; and they bind the object to the current OpenGL context, making it an active object. All subsequent calls that manipulate objects of this kind will refer to the currently bound object. When we are done dealing with an OpenGL object, it is a good practice to unbind it by calling glBind*(..,0) with the parameter 0 instead of an object name.

For example, to create a VertexArray, one needs to call:

```
vertexArrayObj = glGenVertexArrays(); // Get a OGL "name" for a
↪  vertex-array object
glBindVertexArray(vertexArrayObj); // Create a new vertex-array
↪  object with that name and bind it to the current context
```

OpenGL object can be deleted once it is not needed with one of the glDelete* functions. For example:

```
glDeleteVertexArrays(vertexArrayObj);
```

7.3 Loading data into buffers

Loading data into OpenGL buffers is a bit more involved in Java than in C or C++. Since OpenGL is a C rather than Java library, it expects data to be stored in a continuous segment of memory, as they would be stored in C. FloatBuffer class can be used to convert Java's float[] array into the format that is appropriate for OpenGL. For example, to upload vertex positions, one needs to call:

```
FloatBuffer vertex_buffer =
↪  BufferUtils.createFloatBuffer(vertPositions.length);
vertex_buffer.put(vertPositions); // Put the vertex array into
↪  the CPU buffer
vertex_buffer.flip(); // "flip" is used to change the buffer from
↪  write to read mode

int vertex_handle = glGenBuffers(); // Get an OGL name for a
↪  buffer object
glBindBuffer(GL_ARRAY_BUFFER, vertex_handle); // Bring that
↪  buffer object into existence on GPU
glBufferData(GL_ARRAY_BUFFER, vertex_buffer, GL_STATIC_DRAW); //
↪  Load the GPU buffer object with data
```

`GL_ARRAY_BUFFER` specifies a type of object to create. The `flip()` method flips the state from writing to reading so that OpenGL can start reading from our `normal_buffer` object. The function `glBufferData` loads vertex coordinates stored in `vertex_buffer` into the GPU memory.

7.4 Passing vertex data into shaders

The next step is to tell the vertex shader where it can find vertex attributes. You can find the following code in `Mesh.loadDataOntoGPU()`:

```
int shaders_handle = shaders.getHandle();
...
// Get the locations of the "position" vertex attribute variable
↪  in our ShaderProgram
int position_loc = glGetAttribLocation(shaders_handle,
↪  "position");

// If the vertex attribute does not exist, position_loc will be
↪  -1
if (position_loc == -1)
   throw new RuntimeException( "'position' variable not found in
   ↪  the shader file");

// Specifies where the data for "position" variable can be
↪  accessed
```

```
glVertexAttribPointer(position_loc, 3, GL_FLOAT, false, 0, 0);

// Enable that vertex attribute variable
glEnableVertexAttribArray(position_loc);
```

`glGetAttribLocation` function finds the "location" of the input variable `position` in the vertex shader. You can find this variable declared in the first few lines of the vertex shader (in resources/cube_vertex_shader.glsl):

```
#version 140

in vec3 ls_position;    // vertex position in local space
in vec3 ls_normal;      // vertex normal in local space
```

The "location" is in practice an integer index equal to 0 because it is the first input variable in the shader code. However, you should always look-up the location instead of using a fixed constant. If the function returns -1, it means that we mistyped the name and the application should complain about it.

`glVertexAttribPointer` links the currently bound ArrayBuffer to the "position" variable in the shader. Look-up "man glVertexAttribPointer" to check the meaning of other parameters of this function. Finally, `glEnableVertexAttribArray` enables the specified input variable.

7.5 Shader Debugging

Majority of the errors encountered in computer graphics application development, particularly shader programming, are silent run-time errors making the debugging process very frustrating. Some common strategies to debug graphics application include:

- Hypothesis building: Think of what you expected to see as the output and compare with what you got. Try to guess what part of the graphics pipeline might be responsible for the error and check it out.
- Divide and Conquer: Divide your application into small separable parts such as model loading, texture loading, transformations, shader setup, etc. and test them individually.
- Minimal working example: Try commenting out as much code as you can while retaining the error and use simple 3D models, identity transformations and simple shaders. Try to find the bug in this smaller code base.
- GPU debugging tools: You can also explore some GPU debuggers like NVIDIA Nsight, AMD CodeXL, gDebugger, etc.
- Online Forums: Check if someone has encountered a similar problem in the past on the course's Moodle Forum or Stack Overflow.

Here are some common mistakes that you might be making:

- ***I don't see my 3D object.***
  Check the position and scaling of the cube and make sure the camera is outside of the cube.
- ***Only part of the 3D object is displayed.***
  Again check the model transformations. Try changing the near and far plane values in `getProjectionMatrix()` in `Camera.java`.
- ***3D object is in the wrong place.***
  Make sure you are building your transformation matrices wrt OpenGL coordinate system.
- ***3D object is too dark or too bright.***
  Check the distance of light from the model's surface. You don't have to explicitly set any intensity reduction factor with distance in your shading calculation.
- ***I add a light but I don't see any effect***
  Check the direction of your light vector in shading calculation
- ***Robot's arm and body are not rotating in sync***
  Check if you are multiplying your transformation matrices in correct order and uploading the correct matrix to the shader.
- ***Texture is not displayed***
  Are texture files present in the correct folders? Are you using the `texture(...)` function correctly?
- ***Shading doesn't match the tick's images***
  You do need to match exact images as in the tick description as they might have been generated using different light position and colours then yours. But do check if you can generate similar shading results by playing with the camera position.
- ***Reflections look wrong***
  Check the direction of all the vectors involved in the calculation.

For more strategies and tools on debugging, please check out at
http://education.siggraph.org/cgsource/content/debug-gpu-programs