**POLITECHNIKA ŚWIĘTOKRZYSKA**

**KIELCE UNIVERSITY OF TECHNOLOGY**

**Faculty of Electrical Engineering, Automatic Control and Computer Science**

Authors: Renat Lobanov and Oleksandr Lastovets

Study Course: Computer Science

Course: Basics of the computer graphics

Group: EN11A

First-Cycle Studies

Year: 2024

**Documentation of the 3D game engine**

**Main**

| def main():  engine = Engine(1280, 800, "3D", False, 6000)  engine.initialize()  engine.set\_projection(60.0, 1280.0 / 800.0, 0.1, 100.0, 1)  engine.main\_loop()  engine.terminate()   if \_\_name\_\_ == "\_\_main\_\_":  main() |
| --- |

The main function initializes the application by creating an Engine object with specific parameters, including window size (1280x800), title ("3D"), no fullscreen mode, and a refresh rate of 6000. The engine is then initialized with initialize(), and the projection matrix is set using set\_projection(), configuring the field of view, aspect ratio, and clipping planes. The rendering loop starts with main\_loop(), and after it ends, the engine is terminated with terminate().

**Engine**

| class Engine:  def \_\_init\_\_(self, width, height, title, fullscreen=False, fps=60):  if not glfw.init():  raise Exception("GLFW could not be initialized!")  self.width = width  self.height = height  self.title = title  self.fullscreen = fullscreen  self.window = None  self.background\_color = [0.1, 0.2, 0.3, 1.0]  self.frameCount = 0  self.previousTime = glfw.get\_time()  self.projection\_matrix = glm.mat4(1.0)  self.fps = fps  self.frame\_duration = 1.0 / fps  self.window\_should\_close = False  self.camera = Camera(position=[0.0, 0.0, 3.0], target=[0.0, 0.0, 0.0], up\_vector=[0.0, 1.0, 0.0])   def initialize(self):  self.window = Window(self.width, self.height, self.title, self.fullscreen, None)  self.window.setup()  glEnable(GL\_DEPTH\_TEST)   def set\_projection(self, fov, aspect, zNear, zFar, type):  if type == 1:  self.projection\_matrix = glm.perspective(glm.radians(fov), aspect, zNear, zFar)  else:  self.projection\_matrix = glm.ortho(-aspect, aspect, -1.0, 1.0, zNear, zFar)   def main\_loop(self):  while not glfw.window\_should\_close(self.window.getWindow()):  width, height = glfw.get\_window\_size(self.window.getWindow())  glViewport(0, 0, width, height)  currentTime = glfw.get\_time()  self.frameCount += 1  if (currentTime - self.previousTime) >= 1.0:  glfw.set\_window\_title(self.window.getWindow(), f"FPS: {self.frameCount}")  self.frameCount = 0  self.previousTime = currentTime   glClearColor(\*self.background\_color)  glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT)   view = self.camera.view\_matrix  MVP\_half = self.projection\_matrix \* view   cube = Cube(vertex\_shader\_source, fragment\_shader\_source)  textured\_cube = TexturedCube(texture\_vertex\_shader, texture\_fragment\_shader, "textures/wood.png")  textured\_sphere = TexturedSphere(texture\_vertex\_shader, texture\_fragment\_shader, "textures/earth.jpg", radius=1.0, sectors=32, stacks=16)   cube.translate(0.0, 0.0)  cube.rotate(0.0, 0.0, 0.0)  cube.scale(1.0)   textured\_cube.translate(2.0, 2.0)  textured\_cube.rotate(0.0, 0.0, 0.0)  textured\_cube.scale(1.0)   textured\_sphere.translate(-2.0, -2.0)  textured\_sphere.rotate(0.0, 0.0, 0.0)  textured\_sphere.scale(1.0)   textured\_cube.draw(self.projection\_matrix, view, self.camera.get\_position(), [2.0, 2.0, 2.0])  textured\_sphere.draw(self.projection\_matrix, view, self.camera.get\_position(), [2.0, 2.0, 2.0])  cube.draw(self.projection\_matrix, view, self.camera.get\_position(), [2.0, 2.0, 2.0])   glfw.swap\_buffers(self.window.getWindow())  glfw.poll\_events()   def terminate(self):  glfw.terminate() |
| --- |

Engine class that sets up window creation, handles rendering, and processes user input to manipulate objects in a 3D scene. It includes functionalities for object transformations (translation, rotation, scaling) and rendering using shaders. The main loop clears the screen, updates object transformations, and renders objects like cubes, spheres, and triangles. User input controls camera movement and transformation modes for different objects. The program also manages keyboard and mouse events for background color changes and object interactions. The terminate method ensures proper cleanup and window closure.

**Window**

| class Window:  def \_\_init\_\_(self, W, H, title, monitor=False, share=False):  self.W = W  self.H = H  self.title = title  self.monitor = monitor  self.share = share  self.window = None   def setup(self):  if self.monitor:   primary\_monitor = glfw.get\_primary\_monitor()  video\_mode = glfw.get\_video\_mode(primary\_monitor)  self.W, self.H = video\_mode.size.width, video\_mode.size.height  self.window = glfw.create\_window(int(self.W), int(self.H), self.title, primary\_monitor, self.share)  else:  self.window = glfw.create\_window(int(self.W), int(self.H), self.title, None, self.share)  if not self.window:  glfw.terminate()  raise Exception("GLFW window could not be created!")  glfw.make\_context\_current(self.window)   def getWindow(self):  return self.window |
| --- |

The Window class handles the creation and setup of a GLFW window for OpenGL rendering. It takes parameters for the window's width, height, title, and options for fullscreen and window sharing. In the setup method, if fullscreen mode is enabled, the window is created with the primary monitor's resolution. Otherwise, it uses the specified width and height. The window is created with GLFW, and context is set for OpenGL operations. If the window creation fails, an exception is raised.

**GameObject**

| class OpenGLObject(ShapeObject, UpdataleObject):  def \_\_init\_\_(  self,  vertices,  vertex\_shader\_source,  fragment\_shader\_source,  draw\_mode=GL\_TRIANGLES,  ):  ShapeObject.\_\_init\_\_(  self,  vertices,  vertex\_shader\_source,  fragment\_shader\_source,  draw\_mode=GL\_TRIANGLES,  )  UpdataleObject.\_\_init\_\_(self) |
| --- |

The OpenGLObject class inherits from ShapeObject and UpdatableObject, combining shape rendering and update functionality. It initializes a 3D object with vertex data and shader sources for rendering using OpenGL. The class provides the ability to draw objects with different transformation modes (e.g., translation, rotation, scaling) and supports updates to the object’s properties over time. The object is rendered using the specified drawing mode, defaulting to GL\_TRIANGLES.

**DrawableObject**

| class DrawableObject:  def \_\_init\_\_(self, vertices, vertex\_shader\_source, fragment\_shader\_source, draw\_mode=GL\_TRIANGLES):  self.vertices = vertices  self.vertex\_shader\_source = vertex\_shader\_source  self.fragment\_shader\_source = fragment\_shader\_source  self.draw\_mode = draw\_mode  self.shader\_program = self.\_create\_shader\_program()  self.\_setup\_buffers()   def \_setup\_buffers(self):  # Setup vertex buffers and attributes  pass   def \_compile\_shader(self, shader\_type, source):  # Compile shader code  pass   def \_create\_shader\_program(self):  # Create shader program from vertex and fragment shaders  pass   def draw(self, projection\_matrix, view):  glUseProgram(self.shader\_program)  light\_pos\_location = glGetUniformLocation(self.shader\_program, "lightPos")  light\_pos = [1.0, 1.0, 1.0]  glUniform3f(light\_pos\_location, \*light\_pos)  MVP = projection\_matrix \* view \* self.trans  mvp\_location = glGetUniformLocation(self.shader\_program, "MVP")  glUniformMatrix4fv(mvp\_location, 1, GL\_FALSE, glm.value\_ptr(MVP))  glBindVertexArray(self.VAO)  glDrawArrays(self.draw\_mode, 0, len(self.vertices) // 3)   glBindVertexArray(0)   def \_\_del\_\_(self):  # Cleanup buffers and shaders  pass |
| --- |

The DrawableObject class is responsible for handling the creation and rendering of graphical objects in OpenGL. It sets up the necessary OpenGL buffers and shaders for rendering an object with vertex data, applies transformations using a Model-View-Projection (MVP) matrix, and draws the object using the specified drawing mode. The class includes methods for compiling shaders, linking them into a program, and rendering the object with appropriate transformations.

**TransformaleObject**

| class TransformableObject:  def \_\_init\_\_(self):  self.trans = glm.mat4(1.0)   def scale(self, scale):  scale\_matrix = glm.scale(glm.mat4(1.0), glm.vec3(scale, scale, scale))  self.trans \*= scale\_matrix   def rotate(self, angle\_x, angle\_y, angle\_z):  self.trans = glm.rotate(self.trans, glm.radians(angle\_x), glm.vec3(1.0, 0.0, 0.0))  self.trans = glm.rotate(self.trans, glm.radians(angle\_y), glm.vec3(0.0, 1.0, 0.0))  self.trans = glm.rotate(self.trans, glm.radians(angle\_z), glm.vec3(0.0, 0.0, 1.0))   def translate(self, x, y):  self.trans = glm.translate(self.trans, glm.vec3(x, y, 0)) |
| --- |

The TransformableObject class handles basic transformations of objects, including scaling, rotation, and translation. It maintains a transformation matrix (trans), initialized as an identity matrix. The scale method scales the object by the provided factor in all three dimensions. The rotate method applies rotations around the X, Y, and Z axes in sequence. The translate method moves the object in the X and Y directions while keeping the Z value fixed.

**ShapeObject**

| class ShapeObject(TransformableObject, DrawableObject):  def \_\_init\_\_(  self,  vertices,  vertex\_shader\_source,  fragment\_shader\_source,  draw\_mode=GL\_TRIANGLES,  ):  TransformableObject.\_\_init\_\_(self)  DrawableObject.\_\_init\_\_(  self,  vertices,  vertex\_shader\_source,  fragment\_shader\_source,  draw\_mode=GL\_TRIANGLES,  ) |
| --- |

The ShapeObject class combines the functionalities of transformation (from TransformableObject) and rendering (from DrawableObject). It initializes with vertex data, shaders, and the drawing mode. The class inherits methods for object transformations like scaling, rotation, and translation, and also has the ability to render the object using shaders in OpenGL. The ShapeObject is designed to handle both the geometric transformations and the graphical rendering of objects.

**Primitives**

| class Triangle(OpenGLObject):  def \_\_init\_\_(self, vertices, vertex\_shader\_source, fragment\_shader\_source):  super().\_\_init\_\_(vertices, vertex\_shader\_source, fragment\_shader\_source, draw\_mode=GL\_TRIANGLES) |
| --- |

Example of Triangle class.

**Line**:

**Rendering Mode**: Uses GL\_LINES, which draws a straight line between the two vertices.

**Initialization**: Accepts two points (start and end), then passes them as vertices to the OpenGLObject constructor, specifying GL\_LINES as the draw mode.

**Pixel**:

**Rendering Mode**: Uses GL\_POINTS, which renders individual points, often used for particle systems or debugging.

**Initialization**: Accepts a position as a point in 3D space, which is passed as a single vertex to the OpenGLObject constructor with the GL\_POINTS mode.

**Triangle\_fans**:

**Rendering Mode**: Uses GL\_TRIANGLE\_FAN, where the first vertex defines the center and subsequent vertices create triangles radiating out from that center.

**Initialization**: Accepts a list of points (vertices) that define the fan, which is passed to the OpenGLObject constructor.

**Custom Draw Method**: Overrides the draw() method to use the GL\_TRIANGLE\_FAN mode for rendering the vertices.

**Triangle\_strip**:

**Rendering Mode**: Uses GL\_TRIANGLE\_STRIP, which draws a connected sequence of triangles based on the provided vertices.

**Initialization**: Accepts a list of points (vertices) that define the strip, which is passed to the OpenGLObject constructor.

**Custom Draw Method**: Overrides the draw() method to use the GL\_TRIANGLE\_STRIP mode for rendering the vertices.

**Triangle**:

**Rendering Mode**: Uses GL\_TRIANGLES, where each set of three consecutive vertices forms an individual triangle.

**Initialization**: Accepts the vertices of the triangle and passes them to the OpenGLObject constructor, using the GL\_TRIANGLES draw mode.

**Line\_loop**:

**Rendering Mode**: Uses GL\_LINE\_LOOP, which draws a connected sequence of lines that forms a loop.

**Initialization**: Takes a list of points and passes them to the OpenGLObject constructor using the GL\_LINE\_LOOP mode.

**Line\_stripe**:

**Rendering Mode**: Uses GL\_LINE\_STRIP, which renders a series of connected line segments.

**Initialization**: Takes a list of points and passes them to the OpenGLObject constructor using the GL\_LINE\_STRIP mode.

**Camera**

| class Camera:  def \_\_init\_\_(self, position, target, up\_vector):  self.position = glm.vec3(position)   self.target = glm.vec3(target)  self.up = glm.vec3(up\_vector)    self.view\_matrix = glm.mat4(1.0)  self.projection\_matrix = glm.mat4(1.0)    self.update\_view\_matrix()   def update\_view\_matrix(self):  self.view\_matrix = glm.lookAt(self.position, self.target, self.up)   def set\_projection(self, fov, aspect\_ratio, near, far):  self.projection\_matrix = glm.perspective(glm.radians(fov), aspect\_ratio, near, far)   def move(self, delta\_position):  self.position += glm.vec3(delta\_position)  self.target += glm.vec3(delta\_position)  self.update\_view\_matrix()  def get\_position(self):  return self.position |
| --- |

The Camera class handles the position, target, and orientation of a 3D camera. It updates the view and projection matrices, allowing for camera movement and perspective adjustments. The class enables real-time transformations of the scene from the camera’s viewpoint.

**Cube**

| class Cube(OpenGLObject):  def \_\_init\_\_(self, vertex\_shader\_source, fragment\_shader\_source):  self.trans = glm.mat4(1.0)  self.vertices = np.array([###], dtype=np.float32)   self.indices = np.array([###], dtype=np.uint32)   self.colors = np.array(  [###],  dtype=np.float32,  )   self.normals = np.array(  [###],  dtype=np.float32,  )   super().\_\_init\_\_(self.vertices, vertex\_shader\_source, fragment\_shader\_source)   self.shader\_program = super().\_create\_shader\_program()   self.EBO = glGenBuffers(1)  glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, self.EBO)  glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, self.indices.nbytes, self.indices, GL\_STATIC\_DRAW)   glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0)   def \_setup\_buffers(self):  # Setup buffers and attributes    def draw(self, projection\_matrix, view, camera\_position, light\_pos):  glUseProgram(self.shader\_program)   light\_pos = light\_pos  light\_color = [1.0, 1.0, 1.0]  object\_color = [0.8, 0.1, 0.1]  view\_pos = camera\_position   glUniform3f(glGetUniformLocation(self.shader\_program, "lightPos"), \*light\_pos)  glUniform3f(glGetUniformLocation(self.shader\_program, "lightColor"), \*light\_color)  glUniform3f(glGetUniformLocation(self.shader\_program, "objectColor"), \*object\_color)  glUniform3f(glGetUniformLocation(self.shader\_program, "viewPos"), \*view\_pos)   MVP = projection\_matrix \* view \* self.trans  mvp\_location = glGetUniformLocation(self.shader\_program, "MVP")  model\_location = glGetUniformLocation(self.shader\_program, "model")   glUniformMatrix4fv(mvp\_location, 1, GL\_FALSE, glm.value\_ptr(MVP))  glUniformMatrix4fv(model\_location, 1, GL\_FALSE, glm.value\_ptr(self.trans))   glBindVertexArray(self.VAO)  glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, self.EBO)  glDrawElements(GL\_TRIANGLES, len(self.indices), GL\_UNSIGNED\_INT, None)   glBindVertexArray(0)  glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0)   def \_\_del\_\_(self):  # Cleanup buffers and shaders |
| --- |

The Cube class is designed to represent a 3D cube with transformations and graphical rendering in OpenGL. It handles the initialization of vertex data, shader program creation, and drawing of the cube, all while allowing for transformations such as scaling, rotation, and translation. The cube’s vertices, colors, and normals are defined for proper rendering, and the cube can be manipulated.

**BitmapHandler**

| class BitmapHandler:  def \_\_init\_\_(self):  self.textures = {}   def load\_texture(self, file\_path):  if file\_path in self.textures:  return self.textures[file\_path]   try:  image = Image.open(file\_path)  if image.mode != 'RGBA':  image = image.convert('RGBA')  img\_data = np.array(image)   texture\_id = glGenTextures(1)  glBindTexture(GL\_TEXTURE\_2D, texture\_id)   glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT)  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT)  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR)  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR)   glTexImage2D(  GL\_TEXTURE\_2D,  0,  GL\_RGBA,  image.width,  image.height,  0,  GL\_RGBA,  GL\_UNSIGNED\_BYTE,  img\_data  )  glGenerateMipmap(GL\_TEXTURE\_2D)  self.textures[file\_path] = texture\_id  image.close()  return texture\_id  except Exception as e:  print(f"Error loading texture {file\_path}: {str(e)}")  return None   def bind\_texture(self, texture\_id, texture\_unit=GL\_TEXTURE0):  glActiveTexture(texture\_unit)  glBindTexture(GL\_TEXTURE\_2D, texture\_id)   def delete\_texture(self, texture\_id):  if texture\_id:  glDeleteTextures(1, [texture\_id])  for path, tid in list(self.textures.items()):  if tid == texture\_id:  del self.textures[path]   def cleanup(self):  for texture\_id in self.textures.values():  glDeleteTextures(1, [texture\_id])  self.textures.clear()   def \_\_del\_\_(self):  self.cleanup() |
| --- |

The BitmapHandler class simplifies texture management in OpenGL by handling the loading, binding, and deletion of textures. It loads textures from image files, converting them to the appropriate format for OpenGL. Textures are cached to avoid redundant loading, improving performance. The bind\_texture method allows easy binding of textures to OpenGL units, and the delete\_texture and cleanup methods ensure proper resource cleanup when textures are no longer needed.

**LightCube**

| import ctypes class LightCube(OpenGLObject):  def \_\_init\_\_(self, vertex\_shader\_source, fragment\_shader\_source):  self.trans = glm.mat4(1.0)  scale = 0.2  self.vertices = np.array([###], dtype=np.float32)  super().\_\_init\_\_(self.vertices, vertex\_shader\_source, fragment\_shader\_source)   def \_setup\_buffers(self):  # Setup buffers and attributes   def draw(self, projection\_matrix, view, light\_pos):  glUseProgram(self.shader\_program)    self.trans = glm.mat4(1.0)   self.trans = glm.translate(self.trans, glm.vec3(light\_pos[0], light\_pos[1], light\_pos[2]))    MVP = projection\_matrix \* view \* self.trans  mvp\_location = glGetUniformLocation(self.shader\_program, "MVP")    glUniformMatrix4fv(mvp\_location, 1, GL\_FALSE, glm.value\_ptr(MVP))    glBindVertexArray(self.VAO)  glDrawArrays(GL\_TRIANGLES, 0, 36)  glBindVertexArray(0)   def \_\_del\_\_(self):  # Cleanup buffers and shaders  def scale(self, scale):  scale\_matrix = glm.scale(glm.mat4(1.0), glm.vec3(scale, scale, scale))  self.trans \*= scale\_matrix   def rotate(self, angle\_x, angle\_y, angle\_z):  self.trans = glm.rotate(  self.trans, glm.radians(angle\_x), glm.vec3(1.0, 0.0, 0.0)  )  self.trans = glm.rotate(  self.trans, glm.radians(angle\_y), glm.vec3(0.0, 1.0, 0.0)  )  self.trans = glm.rotate(  self.trans, glm.radians(angle\_z), glm.vec3(0.0, 0.0, 1.0)  )   def translate(self, x, y):  self.trans = glm.translate(self.trans, glm.vec3(x, y, 0)) |
| --- |

The LightCube class represents a small cube used as a light source in 3D space. It defines the cube's vertices and handles transformations like scaling, rotation, and translation. The cube is rendered using OpenGL by setting up buffers and drawing with a model-view-projection (MVP) matrix. It also includes methods for cleanup, ensuring proper resource management. The cube’s position is controlled through the light\_pos parameter.

**TexturedCube**

| class TexturedCube(OpenGLObject):  def \_\_init\_\_(self, vertex\_shader\_source, fragment\_shader\_source, texture\_path):  self.trans = glm.mat4(1.0)  self.vertices = np.array([###], dtype=np.float32)  self.indices = np.array([###], dtype=np.uint32)  super().\_\_init\_\_(self.vertices, vertex\_shader\_source, fragment\_shader\_source)  self.EBO = glGenBuffers(1)  glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, self.EBO)  glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, self.indices.nbytes, self.indices, GL\_STATIC\_DRAW)  self.bitmap\_handler = BitmapHandler()  self.texture = self.bitmap\_handler.load\_texture(texture\_path)   def \_setup\_buffers(self):  # Setup buffers and attributes   def draw(self, projection\_matrix, view, camera\_position, light\_pos):  glUseProgram(self.shader\_program)  self.bitmap\_handler.bind\_texture(self.texture, GL\_TEXTURE0)  glUniform1i(glGetUniformLocation(self.shader\_program, "texture1"), 0)  light\_pos = [1.2, 1.0, 2.0]  light\_color = [1.0, 1.0, 1.0]  view\_pos = [0.0, 0.0, 3.0]  glUniform3f(glGetUniformLocation(self.shader\_program, "lightPos"), \*light\_pos)  glUniform3f(glGetUniformLocation(self.shader\_program, "lightColor"), \*light\_color)  glUniform3f(glGetUniformLocation(self.shader\_program, "viewPos"), \*view\_pos)  MVP = projection\_matrix \* view \* self.trans  mvp\_location = glGetUniformLocation(self.shader\_program, "MVP")  model\_location = glGetUniformLocation(self.shader\_program, "model")  glUniformMatrix4fv(mvp\_location, 1, GL\_FALSE, glm.value\_ptr(MVP))  glUniformMatrix4fv(model\_location, 1, GL\_FALSE, glm.value\_ptr(self.trans))  glBindVertexArray(self.VAO)  glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, self.EBO)  glDrawElements(GL\_TRIANGLES, len(self.indices), GL\_UNSIGNED\_INT, None)  glBindVertexArray(0)  glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0)   def \_\_del\_\_(self):  # Cleanup buffers and shaders |
| --- |

The TexturedCube class represents a cube with a texture applied to it. It defines the cube's vertices, texture coordinates, and indices for rendering with OpenGL. Upon initialization, the class loads a texture using BitmapHandler, binds the texture to the shader, and sets up the cube's buffers for efficient drawing. The draw method renders the cube by applying transformations (scaling, rotation, translation), using a model-view-projection matrix. It also manages the cube’s lighting, such as light position and color

**TexturedSphere**

| class TexturedSphere(OpenGLObject):  def \_\_init\_\_(self, vertex\_shader\_source, fragment\_shader\_source, texture\_path, radius=1.0, sectors=32, stacks=16):  self.radius, self.sectors, self.stacks = radius, sectors, stacks  self.trans = glm.mat4(1.0)  self.vertices, self.indices = [], []   for i in range(stacks + 1):  phi = (i / stacks) \* math.pi  for j in range(sectors + 1):  theta = (j / sectors) \* 2 \* math.pi  x, y, z = math.cos(theta) \* math.sin(phi), math.cos(phi), math.sin(theta) \* math.sin(phi)  self.vertices.extend([self.radius \* x, self.radius \* y, self.radius \* z, x, y, z, j / sectors, i / stacks])    for i in range(stacks):  for j in range(sectors):  first, second = (i \* (sectors + 1)) + j, (i \* (sectors + 1)) + j + 1  self.indices.extend([first, second, first + sectors + 1, second, second + sectors + 1, first + sectors + 1])   self.vertices = np.array(self.vertices, dtype=np.float32)  self.indices = np.array(self.indices, dtype=np.uint32)  super().\_\_init\_\_(self.vertices, vertex\_shader\_source, fragment\_shader\_source)   self.bitmap\_handler = BitmapHandler()  self.texture = self.bitmap\_handler.load\_texture(texture\_path)   def \_setup\_buffers(self):  # Setup buffers and attributes  def draw(self, projection\_matrix, view, camera\_position, light\_pos):  glUseProgram(self.shader\_program)  self.bitmap\_handler.bind\_texture(self.texture, GL\_TEXTURE0)  glUniform3f(glGetUniformLocation(self.shader\_program, "lightPos"), \*light\_pos)  glUniform3f(glGetUniformLocation(self.shader\_program, "viewPos"), \*camera\_position)  MVP = projection\_matrix \* view \* self.trans  glUniformMatrix4fv(glGetUniformLocation(self.shader\_program, "MVP"), 1, GL\_FALSE, glm.value\_ptr(MVP))  glUniformMatrix4fv(glGetUniformLocation(self.shader\_program, "model"), 1, GL\_FALSE, glm.value\_ptr(self.trans))  glBindVertexArray(self.VAO)  glDrawElements(GL\_TRIANGLES, len(self.indices), GL\_UNSIGNED\_INT, None)   def \_\_del\_\_(self):  # Cleanup buffers and shaders |
| --- |

The TexturedSphere class creates a textured 3D sphere. It calculates vertices and indices using spherical coordinates, with customizable radius, sectors, and stacks for different levels of detail. The constructor initializes the sphere with vertex and index data and loads a texture. The \_setup\_buffers method binds the necessary buffers for OpenGL rendering. The draw method renders the sphere with lighting and view parameters.



