---

1. config.py — Program Configuration

```
DISPLAY_WIDTH = 500
```

– Sets the width of the window to **500 pixels**.

```
DISPLAY_HEIGHT = 500
```

– Sets the height of the window to **500 pixels**.

```
FPS = 60
```

– The program should attempt to run **60 frames per second**.

```
BACKGROUND_COLOR = (0, 0, 0, 1)  # Black background
```

– An RGBA color (red, green, blue, alpha).
– (0,0,0,1) = **black with full opacity**.
– Used by OpenGL when clearing the screen.

---

2. shader.py — Creating and Compiling Shaders

```
from OpenGL.GL import *
```

– Loads all OpenGL functions, including shader functions.

Vertex Shader Source Code

```
vertex_shader_src = """
#version 330 core
layout (location = 0) in vec2 aPos;
void main()
{
    gl_Position = vec4(aPos, 0.0, 1.0);
}
"""
```

Line-by-line:

- `#version 330 core`
  – Shader follows OpenGL **3.3** rules.
- `layout (location = 0) in vec2 aPos;`
  – Receives a 2D vector from VBO using attribute location **0**.

- `gl_Position = vec4(aPos, 0.0, 1.0);`
  – Converts (x,y) → (x,y,0,1) for the GPU.
  – Determines where the vertex appears on screen.

Fragment Shader Source Code

```
fragment_shader_src = """
#version 330 core
out vec4 FragColor;
void main()
{
    FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
"""
```

- `out vec4 FragColor;`
  – Final pixel color goes into this variable.
- `FragColor = vec4(1,1,1,1);`
  – Outputs **white** color

Shader Compilation Function

```
def compile_shader(shader_type, source):
```

– Function that creates and compiles a shader.

```
    shader = glCreateShader(shader_type)
```

– Creates an empty shader object (vertex or fragment).

```
    glShaderSource(shader, source)
```

– Loads GLSL source code into the shader object.

```
    glCompileShader(shader)
```

– Compiles the shader.

```
    if glGetShaderiv(shader, GL_COMPILE_STATUS) != GL_TRUE:
        raise RuntimeError(glGetShaderInfoLog(shader).decode())
```

– If compilation fails, show compilation errors and stop the program.

```
    return shader
```

– Returns the compiled shader object.

Create Shader Program

```
def create_shader_program():
```

– Combines vertex and fragment shaders into one GPU program.

```
    vertex_shader = compile_shader(GL_VERTEX_SHADER, vertex_shader_src)
```

– Compiles vertex shader.

```
    fragment_shader = compile_shader(GL_FRAGMENT_SHADER, fragment_shader_src)
```

– Compiles fragment shader.

```
    shader_program = glCreateProgram()
```

– Creates an empty shader program.

```
    glAttachShader(shader_program, vertex_shader)
    glAttachShader(shader_program, fragment_shader)
```

– Attaches both shaders to the program.

```
    glLinkProgram(shader_program)
```

– Links them so they work together.

```
    if glGetProgramiv(shader_program, GL_LINK_STATUS) != GL_TRUE:
        raise RuntimeError(glGetProgramInfoLog(shader_program).decode())
```

– Checks for linking errors.

```
    glDeleteShader(vertex_shader)
    glDeleteShader(fragment_shader)
```

– Deletes shaders (they are already stored inside the linked program).

```
    return shader_program
```

– Returns the complete shader program.

---

3. square.py — Creating VAO, VBO, and EBO

```
from OpenGL.GL import *
import numpy as np
import ctypes
```

– Imports OpenGL functions, NumPy for arrays, and ctypes for pointer handling.

Create a Square Function

```
def create_square():
```

– Creates all buffers needed to render a square.

Create Vertex Data

```
    square_vertices = np.array([
        -0.5,  0.5,  # Top-left
        -0.5, -0.5,  # Bottom-left
         0.5, -0.5,  # Bottom-right
         0.5,  0.5   # Top-right
    ], dtype=np.float32)
```

– A list of **2D points** forming a square centered on the origin.

Index Data (EBO)

```
    indices = np.array([
        0, 1,
        1, 2,
        2, 3,
        3, 0
    ], dtype=np.uint32)
```

– Tells OpenGL which vertices to connect using **lines**.

Generate VAO, VBO, EBO

```
    VAO = glGenVertexArrays(1)
    VBO = glGenBuffers(1)
    EBO = glGenBuffers(1)
```

– Allocates GPU memory locations for:

- **VAO** (vertex array object)
- **VBO** (vertex buffer object)
- **EBO** (element/index buffer object)

Bind VAO

```
glBindVertexArray(VAO)
```

– Activates the VAO so all following settings belong to it.

Upload Vertex Data

```
glBindBuffer(GL_ARRAY_BUFFER, VBO)
```

– Select the VBO as the active array buffer.

```
glBufferData(GL_ARRAY_BUFFER, square_vertices.nbytes, square_vertices,
GL_STATIC_DRAW)
```

– Uploads vertex data into GPU memory.

Upload Index Data (EBO)

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO)
```

– Activates EBO for currently bound VAO.

```
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.nbytes, indices,
GL_STATIC_DRAW)
```

– Uploads index data.

Describe Vertex Layout

```
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 2 *
ctypes.sizeof(ctypes.c_float), ctypes.c_void_p(0))
```

Explains **how to read the VBO data**:

- `0` → matches `layout(location = 0)` from vertex shader
- `2` → two floats per vertex (x,y)
- `GL_FLOAT` → data type
- `GL_FALSE` → do not normalize
- `2 * sizeof(float)` → stride size (2 floats per vertex)
- `0` → data starts at offset 0

```
glEnableVertexAttribArray(0)
```

– Enables the attribute so shader can read it.

Unbind Buffers and VAO

```
    glBindBuffer(GL_ARRAY_BUFFER, 0)
    glBindVertexArray(0)
```

– Clean up bindings.

Return Objects

```
    return VAO, EBO, len(indices)
```

– Returns:

- The VAO to use for drawing
- The EBO for cleanup
- Number of indices to draw

---

4. main.py — Creating Window & Rendering

```
import pygame
from pygame.locals import *
from OpenGL.GL import *
import config
from shader import create_shader_program
from square import create_square
```

– Imports pygame, OpenGL, configuration, shader creation, and square creator.

main() function

```
def main():
    pygame.init()
```

– Initializes all pygame modules.

```
    display = (config.DISPLAY_WIDTH, config.DISPLAY_HEIGHT)
```

– Window size (500,500).

```
pygame.display.set_mode(display, DOUBLEBUF | OPENGL)
```

– Creates an OpenGL-enabled window using **double buffering**.

OpenGL Setup

```
glClearColor(*config.BACKGROUND_COLOR)
```

– Sets screen clear color (black).

Build Shader Program

```
shader_program = create_shader_program()
```

– Compiles & links vertex + fragment shaders.

Create Square Buffers

```
VAO, EBO, index_count = create_square()
```

– Gets objects needed to render the square.

```
glUseProgram(shader_program)
```

– Activates the shader program.

Game Loop Setup

```
clock = pygame.time.Clock()
running = True
```

– Initializes FPS controller and loop flag.

Main Loop

```
while running:
```

Event Handling

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        running = False
```

– Exits when window is closed.

Rendering

```
glClear(GL_COLOR_BUFFER_BIT)
```

– Clears screen using glClearColor.

```
glUseProgram(shader_program)
```

– Ensure shader is active.

```
glBindVertexArray(VAO)
```

– Use the square's VAO.

```
glDrawElements(GL_LINES, index_count, GL_UNSIGNED_INT, None)
```

– Draws square **using line segments**.

```
glBindVertexArray(0)
```

– Unbind VAO.

Swap Buffers & Control FPS

```
pygame.display.flip()
```

– Shows the newly rendered frame.

```
clock.tick(config.FPS)
```

– Limits frame rate.

Cleanup

```
glDeleteVertexArrays(1, [VAO])
```

```
    glDeleteBuffers(1, [EBO])
    glDeleteProgram(shader_program)
```

– Deletes GPU resources.

```
    pygame.quit()
```

– Closes Pygame.

```
if __name__ == "__main__":
    main()
```

– Runs the program.

What IS GLSL?

GLSL = OpenGL Shading Language
It is the programming language used to write **GPU programs** called **shaders**.

GLSL code runs **inside the video card**, not on the CPU.

Example GLSL vertex shader:

```
#version 330 core
layout (location = 0) in vec2 aPos;

void main()
{
    gl_Position = vec4(aPos, 0.0, 1.0);
}
```

Example GLSL fragment shader:

```
#version 330 core
out vec4 FragColor;

void main()
{
    FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

GLSL is used to:

- transform vertices
- assign colors
- create lighting
- compute shadows
- apply textures
- control rendering effects.

What is a VBO (Vertex Buffer Object)?

A **VBO** is a **buffer stored in the GPU** that contains your **vertex data**.
Examples of data inside a VBO:

- Positions (x, y, z)
- Colors (r, g, b)
- Texture coordinates (u, v)
- Normals for lighting

Purpose:
✔ Sends large batches of vertex data to the GPU efficiently.

Analogy:
A VBO is like a storage box containing all your vertex points.

What is an EBO (Element Buffer Object)?

Also called **Index Buffer Object (IBO)**.
An **EBO** stores **indices** — integers that tell OpenGL how to reuse vertices.

Example:

```
Vertices: v0, v1, v2, v3
Indices: 0, 1, 2, 2, 3, 0
```

Instead of repeating vertices, OpenGL reuses them to form shapes.

Purpose:
✔ Saves memory
✔ Prevents duplicated vertices
✔ Helps draw shapes like squares, cubes, models

Analogy:
EBO = a list of instructions: "use vertex 0, 1, and 2 to draw triangle 1…"

What is a VAO (Vertex Array Object)?

A **VAO** is an object that **stores the configuration** of how vertex data is read.
It remembers:

- Which VBO is active
- How vertex attributes are laid out
  (e.g., "2 floats per position, starting at offset 0")
- Which EBO is connected
- Which attribute belongs to which shader input

Purpose:

✔ Keeps all the VBO/EBO/attribute layout info bundled together

✔ Makes switching between models fast

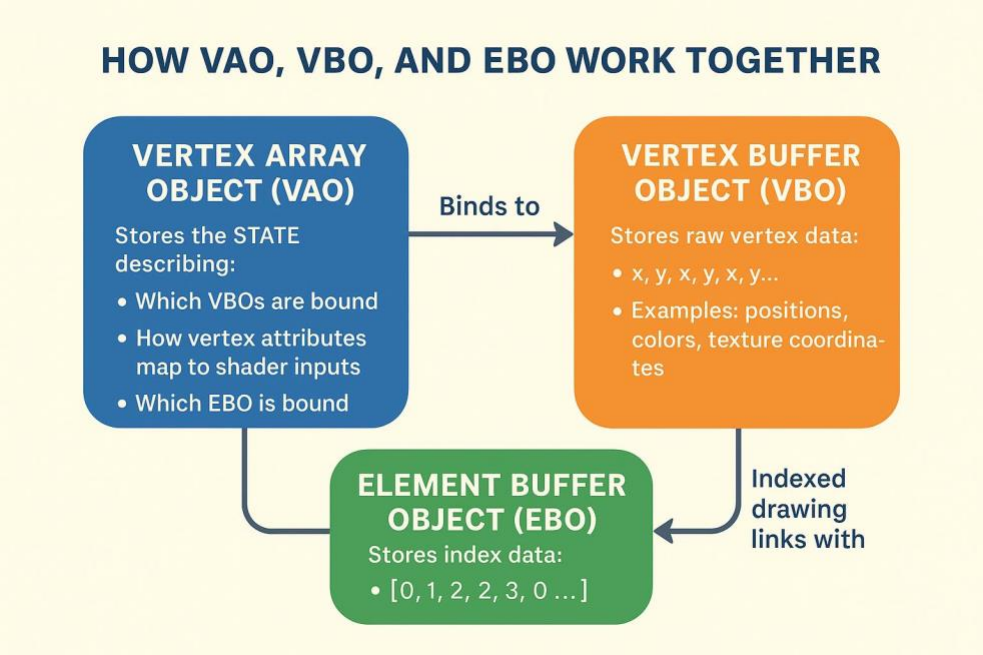
Analogy:

VAO = "recipe card" that tells OpenGL how to read the VBO/EBO data.

You bind a VAO once and OpenGL remembers everything.

VISUAL SUMMARY

```
VAO
 ├── remembers → VBO (vertex data)
 └── remembers → EBO (indices)
```



How They All Connect Together

```
Python/OpenGL code (CPU):
    VAO → remembers layout
    VBO → stores vertex data
    EBO → stores index data

GPU shaders (GLSL):
    Vertex shader → processes each vertex
    Fragment shader → colors each pixel
```