CAP4720 Racing Game

Fall 2023

Chris Jenkins, Alex Cruz, Tyler Kierstein, Marc Cross, Daniel Thew

Sections

i. Technology Used
ii. General Description
iii. Implemented Features
iv. Individual Contributions
v. Results
vi. Video Demonstration

Technology Used

ModernGL

ModernGL is a wrapper for the standard OpenGL graphics API. It essentially condenses much of OpenGL's lower-level functions and makes the whole pipeline much simpler to access and prepare. ModernGL does not affect shaders—it only applies to rendering in OpenGL. ModernGL is also much more intuitive to use with Python due to its use of objects to access features rather than raw access to OpenGL functions.

PyGame

PyGame is the most common framework for game development in Python. It's used to initiate the game loop, set some of the display attributes, and access the user's key inputs for movement and escaping the game window.

PyGLM

PyGLM is a graphics-focused mathematics library designed for Python. It's used to set up vectors and perform common trigonometric functions like normalization and sin/cos.

NumPY

NumPY is a common Python mathematics function library, used to turn datasets into arrays.

PyWavefront

PyWavefront is a Python library used to import .obj files and texture them with .mtl and .png files.

GuiV3

The GUI class provided by Prof. Devkota, used to manually adjust certain features.

CoderSpace Tutorials

This playlist was instrumental in helping design the ModernGL graphics engine.

General Description

Our project is a single-player racing game in which the player drives around a track and attempts to improve on their best time.

Implemented Features

Movement

The user can press W/S for forward/backward movement. Upon moving in a given direction, the user will accelerate until a certain max speed and maintain that speed until the key is released or until the movement key for the opposite direction is pressed, which will cause the user to decelerate in the direction of the key pressed. For instance, if the user holds W, letting go and holding S will decelerate the car to 0, then move the car backwards at an increasing rate of speed. The user can also steer the car with A and S; steering is only available so long as the car is moving, either as a result of a W/S keypress or residual velocity while decelerating. The user can manually adjust maximum speed, rate of acceleration, and steering sensitivity via the Garage GUI.

Lighting

The user can select between a Day and Night setting. The Day setting uses a skybox with a cloudy daytime setting and the source lighting is left at maximum; the Night setting uses a skybox with a dark, starry setting and the source lighting reduced to 40%. Individual objects have their own ambient, diffuse, and specular colors and shadows, but objects do not cast shadows—there are no obstructions to cast shadows, so it was ultimately unnecessary.

Textures

The car's texture can be manually adjusted using a checkbox in the Garage GUI. There are five different colored texture options and an option to show the UV map on the car object. There is also a texture applied to the course, which is not user-adjustable.

Timer

Both the user's current and best times are tracked directly above the user's car via Cube objects and ten different number textures applied to them. The car class tracks the time from its initialization to the time it passes the starting mark of the track, but will only update the user's best time if the user has crossed the track's halfway XZ axes. The timer tracks up to 99.9 seconds, which is more than a reasonable estimate to a user's maximum time given the size of the track and the speed of the car.

Menu

The game uses a simple menu which allows the user to either begin the game or quit. The Start option greatly improves quality of life, as now the user will be prepared to start playing, and thus prepared for when the game begins timing their lap.

Collisions

Our collision handler creates 2D versions of the track (list of rectangles) and car (4 points representing the corners), then checks for any collisions between the points. Upon a collision, the car gets "bounced" backwards and its acceleration is reset to 0.

Individual Contributions

Daniel

I largely followed the CoderSpace tutorial playlist to set up the ModernGL graphics engine, but I had to make adjustments to implement our specific features. I used the GuiV3 file from our class to set up the Garage GUI and created several functions to allow different parts of the engine to access the user's selected values. Selecting a different time of day will trigger a change in an if statement in the Light class to change the level of light, as well as change the path directory for the skybox image folder between a daytime and nighttime option. Changing the car texture does roughly the same thing, applying a different texture id to the user's car object. Changing the car's

speed, acceleration, and steering changes some “constants” in the Camera class, which are then applied to the car’s camera’s position.

The camera class is also used to essentially handle movement. It’s the car’s camera that the user technically moves through the track, and the time cubes above the user’s car have their own cameras that the user simultaneously moves with the car to keep them above the user’s car at all times. The texture on each cube is modified based on a sequence of division and modulus operations applies to the Car class’s current time, then attaching the single-digit int to the path of the texture. When the Car class’s current time at the end of the lap is less than its previous best time (or the best time was 0.0, meaning none had been recorded yet), the current time is applied to the best time. The timer resets after each lap, no matter what.

The objects in the scene were placed after trial and error. Fortunately, the car only moves along the XZ axis, so the only Y axis adjustments were made to align the bottom of the car with the top of the track, as well as placing the time cubes at a good distance above the user’s car.

The engine uses a few different auxiliary classes, Mesh and SceneRenderer. These are simply used to handle several components of the overall program. The Mesh handles the app’s VAOs and textures, while the SceneRenderer communicates between the Main and the Scene to render each part of the scene from the main function. The ShaderProgram makes opening and applying shaders to rendering different objects easier and more modularized.

The VAO class ascribes the appropriate shader programs to each object, as well as the appropriate VBO. The VBO class tells the shader how the object’s vertices are formatted and what attributes it has, gets the vertex data, and returns it as an array. This is necessary in rendering each object as we know from our lectures on the OpenGL pipeline and VAOs/VBOs.

There are three main shader sets: default, car, and advanced_skybox. The default shaders are applied to the track and the time cubes, the car shaders are applied to the car, and the advanced_skybox shaders are applied to the skybox. The default vert shader just gets the UV from the texture cords, the fragPos, the normal and gl_Position given the standard MVP (model, view, projection) matrices. It also gets some simple shadow coordinates. The default frag shader applies these, as well as the scene’s Light, texture, camera position, and shadow map to apply ambient, diffuse, and specular light to each object in the scene. The car vert and frag shaders are

almost identical—they were originally designed to be significantly different dependent on shadow-casting and differing shininess, but because these features were scraped, the separate files are vestigial. Lastly, the skybox's vert shader only applies the gl_Position and clipping coordinates, while its frag shader maps the texture.

I also wrote this report to the rubric's specifications, created the PowerPoint, and created and uploaded the video demo to YouTube.

Marc

For the final project, I developed the main menu and integrated that with the racing code. This required working with mouse handler events and fonts in PyGame. Positioning of the words and the rectangles needed for the buttons was difficult at first, but after a bit of research I understood how to properly use the center of a background object to align text to its center rather than having to hard-code values and hope for the best. I also was able to find out how PyGame deals with limiting FPS to a maximum value.

Alex

For our final project, we did a racing game and my job was to make it so that the user could edit how the car looked. We have 6 different options that will allow you to change the cars color by using different textures instead of implementing a color picker. The textures can be switched around by using the radio buttons on the GUI that is available as soon as the program starts.

Tyler

I was in charge of creating the models for the project. This also included trying to get the UVs of the models to match any textures we wanted to apply to our objects. With the help of blender, I could clean up the UVs to better display the textures of our objects.

Chris

For the project I implemented collision between the vehicle and the walls of the track. This involved creating a Collider class that handles collision between the car and the walls of the track as well as implementing this class in the movement section of the project. This proved difficult due to the way our project handled the position of the car relative to the camera as well as figuring out how to find the coordinates of the walls to use for collision.

Results

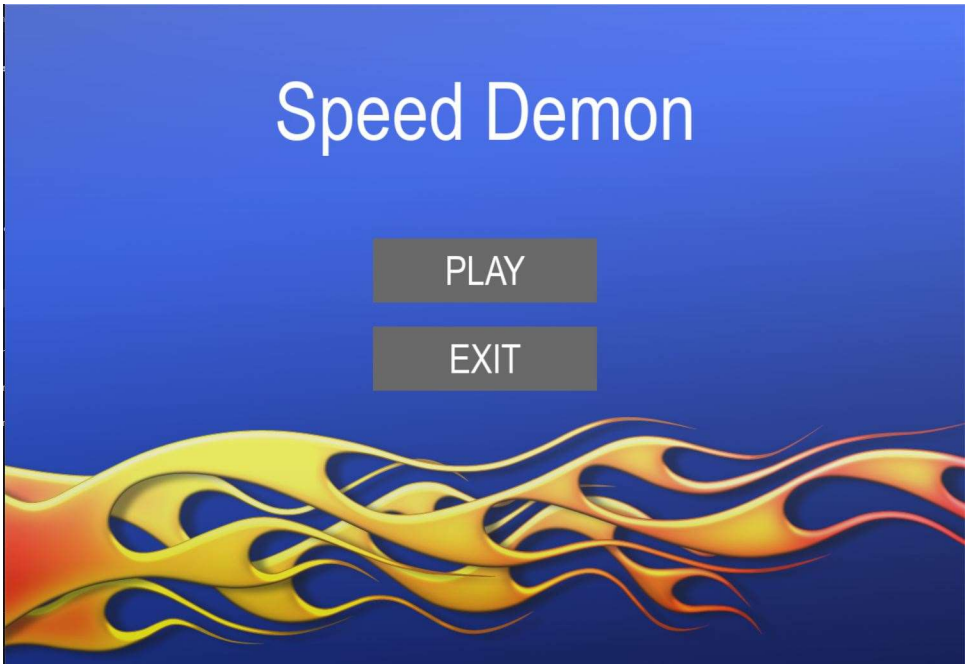

Figure 1 - The menu before starting to play. Users can either begin or quit the game.

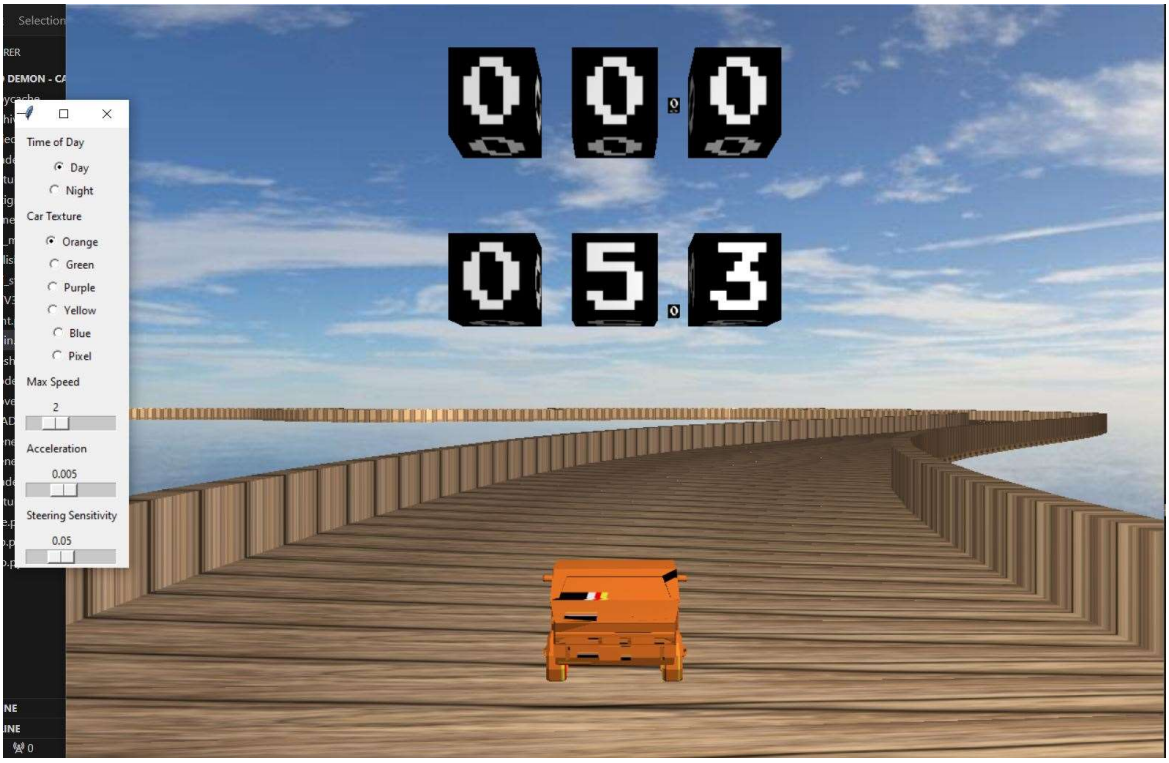

Figure 2 - How the game looks upon loading. The user can change certain settings using the left GUI. The current lap time is displayed directly overhead, and the best time is displayed above that.

Figure 3 - The scene after the user modifies parameters; after completing a lap, the Best Time has been updated.

Video demonstration