**Introduction:** This project is a 3D graphics simulation of a local fair , created using OpenGL and GLFW , GLAD and GLM . The simulation features various stalls, rides and attractions, including a Ferris wheel, a ride and other interactive elements. The project utilizes various techniques such as lighting and texturing to create a realistic and visually appealing environment. The simulation also includes interactive elements, allowing users to navigate and explore the fair using a variety of input keys. The project is divided into several components, including the creation of 3D models, implementation of graphics rendering, and development of interactive elements.

**Features:** Features of the Local Fair Simulation Project ,

∙ 3D Graphics Rendering: The project uses OpenGL and GLFW to create a 3D graphics rendering of local fair , complete with detailed models of rides, attractions, and scenery. ∙ Interactive Navigation: Users can navigate the fair using a variety of input keys. ∙ Realistic Lighting and Shading: The project includes realistic lighting and shading effects, creating a immersive and engaging environment.

∙ Texturing and Materials: The project uses texturing and materials to create realistic and detailed surfaces for the 3D models.

∙ Animated Rides and Attractions: The project includes animated rides and attractions, such as a Ferris wheel and circular rides which can be controlled by the user.

**Implementation:**

****Fig : Local Fair

**Type 1 Stalls :**

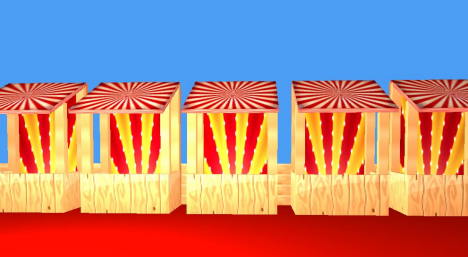
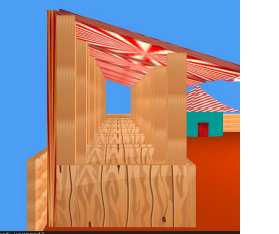
****

Fig : Type 1 stalls

**Different views of type 1 stalls**

****Fig: Front view of stall Fig: Side view of stall

Here is the step-by-step procedure for the stall1:

Step 1: Initialize Variables and matrices for transformation

Step 2: Draw the Base of the Stall

Set the scale matrix to scale the stall base

Set the translation matrix to translate the stall base

Multiply the combined transformation matrix by the scale and translation matrices Draw the base of the stall using the a cube

Step 3: Draw the Back of the Stall

Set the scale matrix to scale the stall back

Set the translation matrix to translate the stall back

Multiply the combined transformation matrix by the scale and translation matrices Draw the back of the stall using the cube

Step 4: Draw the Left Side of the Stall

Set the scale matrix to scale the stall

Set the translation matrix to translate the stall

Multiply the combined transformation matrix by the scale and translation matrices Draw the left side of the stall using the cube

Step 5: Draw the Right Side of the Stall

Set the scale matrix to scale the stall

Set the translation matrix to translate the stall

Multiply the combined transformation matrix by the scale and translation matrices Draw the right side of the stall using the cube

Step 6: Draw the Sticks of the Stall

Set the scale matrix to scale the stick

Set the translation matrix to translate the stick

Multiply the combined transformation matrix by the scale and translation matrices Draw stick using the cube

Repeat this step for each stick

Step 7: Draw the Roof of the Stall

Set the scale matrix to scale the roof

Set the translation matrix to translate the roof

Rotate the roof around the x-axis

Multiply the combined transformation matrix by the rotation matrix, scale matrix, and translation matrix

Draw the roof of the stall using the cube

**Type 2 Stalls :**

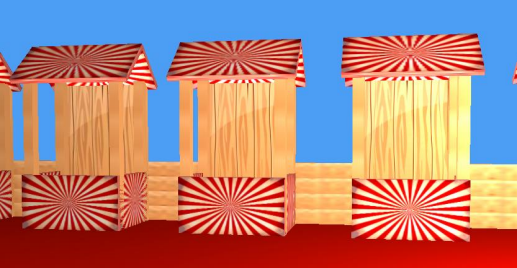
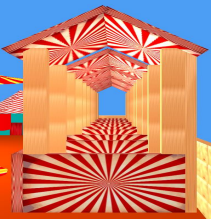
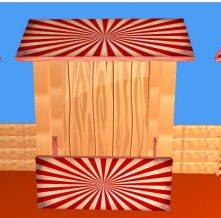
****

Fig : Type 2 Stalls

**Different views of type 1 stalls**

****Fig: Front view of stall Fig: Side view of stall

Here is the step-by-step procedure for the stall2:

It is same as stall1 to the step 6 before the roofs. Here two roof is rotated around x axis one is positive to x axis and one is negative to x axis.

Step 7: Draw the Roof of the Stall

Set the scale matrix to scale the roof

Set the translation matrix to translate the roof

Rotate the roof around the x-axis

Multiply the combined transformation matrix by the rotation matrix, scale matrix, and translation matrix

Draw the roof of the stall using the cube

Step 8: Draw the Second Roof of the Stall

Set the scale matrix to scale the roof

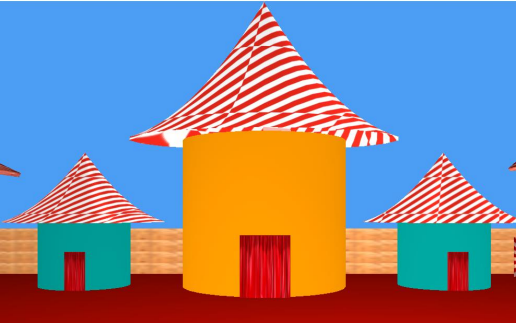
Set the translation matrix to translate the roof

Rotate the roof around the x-axis

Multiply the combined transformation matrix by the rotation matrix, scale matrix, and translation matrix

Draw the second roof of the stall using the cube

**Type 3 Stalls :**

****Fig : Type 3 stalls

These stalls were made with curvy objects. The roofs are made with bezire curve to give them shape like canopies and the base are made with cylinder.

Here is the step-by-step procedure for the Curve class:

Step 1: Initialize Variables

Initialize the control points vector cntrlPoints

Initialize the coordinates vector coordinates

Initialize the normals vector normals

Initialize the indices vector indices

Initialize the vertices vector vertices

Initialize the texture coordinates vector texCoords

Initialize the diffuse map diffuseMap

Initialize the specular map specularMap

Initialize the shininess shininess

Step 2: Constructor

Initialize the fishVAO and bezierVAO variables

Call the hollowBezier function to generate the curve

Set the texture properties

Step 3: Draw Function

Use the lightingShader to draw the curve

Set the model matrix, ambient, diffuse, and specular colors Set the texture properties

Bind the vertex array object fishVAO

Draw the curve using glDrawElements

Step 4: Set Texture Property Function

Set the diffuse map, specular map, and shininess

Step 5: Hollow Bezier Function

Generate the curve using the Bezier curve algorithm

Calculate the coordinates, normals, and texture coordinates Generate the index list of triangles

Create the vertex array object bezierVAO and vertex buffer object bezierVBO Set the attrib arrays with stride and offset

Step 6: Bezier Curve Function

Calculate the x and y coordinates of the curve using the Bezier curve algorithm

Step 7: nCr Function

Calculate the number of combinations of n items taken r at a time

Here is the step-by-step procedure for the drawCylinder function:

Step 1: Initialize Variables

Initialize the lighting shader lightingShader

Initialize the color color of the cylinder

Initialize the model matrix altogether

Step 2: Set Up Model Matrix

Create a new model matrix model and set it to the identity matrix

Multiply the model matrix by the altogether matrix

Set the model matrix in the lighting shader

Step 3: Define Cylinder Parameters

Define the number of segments n for the cylinder

Initialize vectors v and in to store vertex data and indices, respectively

Step 4: Generate Vertex Data

Loop through each segment i and calculate the x, y, and z coordinates of the top and bottom vertices of the segment

Store the vertex data in the v vector, including the color information

Step 5: Generate Index Data

Loop through each segment i and generate the indices for the top and bottom triangles of the segment

Store the index data in the in vector

Step 6: Create Vertex Array Object (VAO) and Vertex Buffer Object (VBO) Generate a new VAO and VBO using glGenVertexArrays and glGenBuffers Bind the VAO and VBO using glBindVertexArray and glBindBuffer

Step 7: Set Up Vertex Attributes

Set up the vertex attribute pointers for the position and color attributes using glVertexAttribPointer

Enable the vertex attributes using glEnableVertexAttribArray

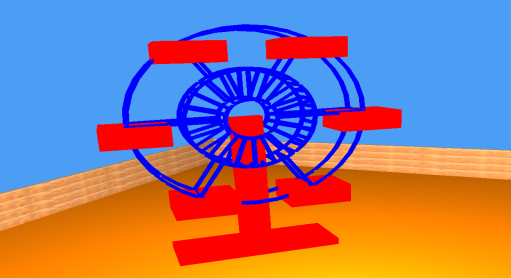
Step 8: Set Up Material Properties

Set the material properties, including the ambient, diffuse, and specular colors, and the shininess value

Step 9: Draw the Cylinder

Draw the cylinder using glDrawElements with the index data stored in the in vector Step 10: Clean Up

Unbind the VAO and delete the VBO and VAO using glDeleteBuffers and glDeleteVertexArrays **Ferris Wheel :**

****Fig : Ferris wheel

Here is the step-by-step procedure for the drawFlywheel function with a focus on the rotation part and the drawCircle part:

Step 1: Initialize Variables

Initialize the lighting shader lightingShaderWithTexture

Initialize the color color of the flywheel

Initialize the model matrix alTogether and alTogether2

Initialize the cube VAO cubeVAO

Step 2: Apply Transformations

Apply translation, rotation, and scaling to the model matrix

Create a new model matrix model and apply transformations to it

Create another model matrix model1 and apply transformations to it

Step 3: Rotate the Flywheel

Check if the flywheel is rotating isFlywheelRotating

If it is, create a rotation matrix rotateFlywheelMatrix using the glm::rotate function Multiply the model matrix with the rotateFlywheelMatrix to get the final model matrix model3 If it's not rotating, set model3 to model

Step 4: Set Up Rotation

Set the rotation angle flywheelRotation to be used in the rotation matrix Use the glm::radians function to convert the rotation angle to radians

Step 5: Draw the Flywheel

Draw the flywheel using the following steps:

Draw circles using the drawCircle function

Draw lines connecting the circles using the drawLine function

Draw seats on the flywheel using the cubeVAO

Step 6: Draw Circles

Draw circles using the drawCircle function with the following parameters:

Center of the circle: (0.0f, -0.4f, 0.0f) for the small circle behind, (0.0f, 0.0f, 0.0f) for the large circle in front, etc.

Radius of the circle: 0.5f for the small circle, 0.9f for the larger circle, etc. Number of segments: 36

Here is some sample code that illustrates the drawCircle part:

// Draw circles

drawCircle(0.0f, -0.4f, 0.0f, 0.5f, 36); // Small circle behind

drawCircle(0.0f, 0.0f, 0.0f, 0.5f, 36); // Large circle in front

drawCircle(0.0f, -0.4f, 0.0f, 0.9f, 36); // Larger circle behind

drawCircle(0.0f, 0.0f, 0.0f, 0.9f, 36); // Larger circle in front

drawCircle(0.0f, -0.4f, 0.0f, 0.2f, 36); // Smallest circle behind

drawCircle(0.0f, 0.0f, 0.0f, 0.2f, 36); // Smallest circle in front

This code draws six circles with different centers and radii to form the flywheel. Step 7: Draw Lines

Draw lines connecting the circles using the drawLine function

Draw lines connecting the smaller circles

Draw lines connecting the larger circles

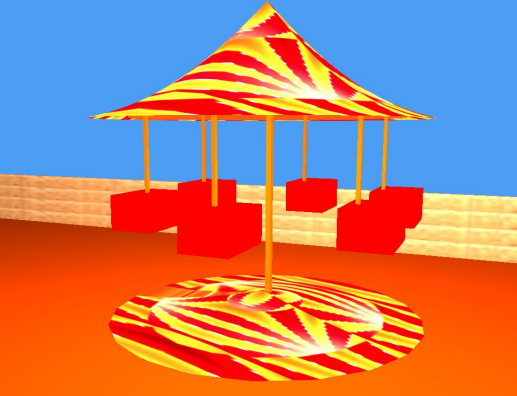
Step 8: Draw Seats

Draw seats on the flywheel using the cubeVAO

Use the model3 matrix to transform the seats

Apply rotation to the seats using the glm::rotate function

**Ride:**

****Fig : Ride

Here is the step-by-step procedure for the drawride function:

Step 1: Initialize Variables

Initialize the lighting shader lightingShaderWithTexture Initialize the color color of the ride Initialize the model matrix alTogether and alTogether2 Initialize the cube VAO cubeVAO

Step 2: Apply Transformations

Apply translation, rotation, and scaling to the model matrix Create a new model matrix model and apply transformations to it Create another model matrix id1 and apply translation to it

Step 3: Draw Canopi

Draw the canopi using the canopi1 function Set the color of the canopi to color Set the model matrix to model

Step 4: Draw Ride

Draw the ride using the cubeVAO Set the model matrix to model \* rideModel Set the color of the ride to glm::vec3(1, 1, 0)

Step 5: Draw circle line

Draw the circle line using the drawCircleflat function Set the model matrix to model Step 6: Animate Ride

Increase the ride's angle by 0.05 degrees every frame Calculate the seat angle using the ride's angle and the angle increment between seats Draw the seats using the cubeVAO Set the model matrix to model \* rideModel Set the color of the seats to glm::vec3(1, 1, 0)

Step 7: Draw Additional Seats

Draw additional seats using the cubeVAO Set the model matrix to model \* rideModel Set the color of the seats to glm::vec3(1, 0, 0) Here is a more detailed explanation of the animation part:

The ride's angle is increased by 0.05 degrees every frame The seat angle is calculated using the ride's angle and the angle increment between seats The seats are drawn using the cubeVAO and the model matrix is set to model \* rideModel The color of the seats is set to glm::vec3(1, 1, 0) Here is some sample code that illustrates the animation part:

static float rideAngle = 0.0f;

**Conclusion**

In conclusion, the project successfully created a 3D simulation of a local fair using OpenGL. The project demonstrated a good understanding of 3D math and OpenGL concepts, and the animation techniques used created a realistic and engaging simulation. However, the project also faced challenges and limitations, and future improvements are suggested to increase the realism, performance, and user engagement of the simulation.