CS405 PROJECT-2 REPORT

MELİH ÇAĞAN ARI

28426

TASK 1

The methodology behind Task 1 involves accommodating non-power-of-two sized textures in WebGL. When the provided image dimensions aren't powers of two, this implementation adjusts the texture parameters to ensure proper handling within WebGL.

When a non-power-of-two texture is detected:

gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);

This parameter ensures that the texture's horizontal (S) wrapping behavior is set to CLAMP_TO_EDGE, preventing any edge wrapping issues.

gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);

Similar to the previous parameter, this ensures the vertical (T) wrapping behavior is set to CLAMP_TOEDGE.

gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);

This sets the method of how the texture is sampled when displayed at a size smaller than its original. In this case, it uses LINEAR filtering.

By adjusting these parameters, the code enables WebGL to handle textures with non-power-of-two dimensions appropriately, allowing for greater flexibility when dealing with images that don't adhere to the traditional power-of-two size requirements.

TASK 2

1. Constructor:

- **Uniform Locations (gl.getUniformLocation):**

this.enableLightingLoc = gl.getUniformLocation(this.prog, 'enableLighting');

this.lightPosLoc = gl.getUniformLocation(this.prog, 'lightPos');

this.ambientLoc = gl.getUniformLocation(this.prog, 'ambient');

this.normalLocation = gl.getUniformLocation(this.prog, 'normal');

this.normalMatrixLocation = gl.getUniformLocation(this.prog, 'v_normal');

These lines retrieve the locations (pointers/handles) of various uniform variables within the shader program (this.prog). These uniforms control aspects of lighting, material properties, and transformations in the vertex and fragment shaders.

- **Buffer Creation (gl.createBuffer):**

this.vertbuffer = gl.createBuffer();

this.texbuffer = gl.createBuffer();

this.normalbuffer = gl.createBuffer();

These lines create buffer objects using gl.createBuffer(). Each buffer (vertbuffer, texbuffer, and normalbuffer) is presumably designated to store specific data for vertices, textures, and normals associated with the rendered object.

this.vertbuffer: Likely used to store vertex position data.

this.texbuffer: Possibly used for texture coordinates.

this.normalbuffer: Intended for storing normal vectors related to the mesh.

After executing this code, the program would have references to the locations of several important uniform variables within the shaders, as well as buffers created to hold vertex, texture, and normal data. These are essential components needed for rendering objects properly using WebGL.

2. SetMesh() Function

- **Buffer Binding (gl.bindBuffer):**

gl.bindBuffer(gl.ARRAY_BUFFER, this.normalbuffer): Binds the buffer object this.normalbuffer to the ARRAY_BUFFER target in WebGL. This buffer will store the normal vector data for the mesh.

- **Data Allocation and Initialization (gl.bufferData):**

gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(normalCoords), gl.STATIC_DRAW);: Initializes the buffer with data for the normal vectors.

gl.ARRAY_BUFFER: Specifies the target buffer to which data will be written (already bound in the previous step).

new Float32Array(normalCoords): Creates a new Float32Array containing the normal vector data (normalCoords). This array type ensures that the data is represented as 32-bit floating-point numbers.

gl.STATIC_DRAW: Indicates that the data provided will be used as static data and is unlikely to change frequently. This hint helps WebGL optimize the data storage and usage.

In essence, this sequence of WebGL commands prepares the this.normalbuffer buffer to hold normal vector data represented as 32-bit floating-point numbers. Once data is loaded into this buffer, it will be accessible for usage in the rendering pipeline to attribute the correct normals to vertices during the rendering process.

3. draw() Function

- **Buffer Binding (gl.bindBuffer):**

gl.bindBuffer(gl.ARRAY_BUFFER, this.normalbuffer);: Binds the buffer this.normalbuffer to the ARRAY_BUFFER target. This buffer likely contains the normal vectors associated with the vertices of the mesh being rendered.

- **Attribute Location Retrieval (gl.getAttribLocation):**

const normalAttrib = gl.getAttribLocation(this.prog, 'normal');: Retrieves the attribute location for the normal vectors within the shader program (this.prog). The attribute named 'normal' in the shader program is expected to receive the normal data for each vertex.

- **Enable Vertex Attribute Array (gl.enableVertexAttribArray):**

gl.enableVertexAttribArray(normalAttrib);: Enables the vertex attribute array at the specified attribute location (normalAttrib), indicating that this attribute will be used in the rendering process.

- **Attribute Pointer Setup (gl.vertexAttribPointer):**

gl.vertexAttribPointer(normalAttrib, 3, gl.FLOAT, false, 0, 0);: Describes the data format and layout of the normal vectors within the buffer:

normalAttrib: The attribute location.

3: Specifies that each normal vector has three components (x, y, z).

gl.FLOAT: The data type of each component is a floating-point number.

false: Indicates whether the data should be normalized. In this case, it's set to false.

0: Represents the stride, which is 0 indicating tightly packed data.

0: Indicates the offset to the first component of the first vertex in the buffer.

This code sequence prepares WebGL to use the normal vectors stored in this.normalbuffer. It sets up the attribute pointer to tell the WebGL pipeline how to interpret and utilize these normal vectors when rendering the associated mesh. The normalAttrib variable stores the location of the normal attribute within the shader program, ensuring that vertex shader attributes and buffer data are correctly synchronized for rendering.

4. Enable Lighting() Function

- **Shader Setup:**

The function starts by ensuring the usage of the designated program (this.prog) for rendering operations, indicating that subsequent operations will apply to this shader program.

- **Uniform Location Retrieval:**

It attempts to retrieve the uniform location associated with the variable enableLighting within the shader program. This uniform likely serves as a flag to enable or disable lighting calculations.

- **Uniform Assignment:**

If the enableLightingLoc is successfully obtained (not null), the function proceeds to update the uniform variable.

gl.uniform1i(enableLightingLoc, show ? 1 : 0);

This line assigns a value (1 for true or 0 for false, based on the show parameter) to the enableLighting uniform. It essentially toggles the lighting on or off within the shader program, allowing for dynamic control over the rendering of illuminated scenes.

- **Error Handling:**

If the uniform location retrieval fails (enableLightingLoc is null), an error message is logged to the console, indicating the failure to locate the uniform responsible for enabling lighting.

Overall, Task 2 calls for the integration of lighting calculations within the WebGL rendering pipeline, using the enableLighting function as a switch to control whether lighting effects should be applied during rendering. This involves setting up appropriate uniforms in the shader program and implementing lighting algorithms to enhance the visual appearance of the rendered scene.

5. setAmbientLight() Function

This function, similar to enableLighting, contributes to the implementation of lighting effects in the WebGL rendering pipeline. It manages the ambient light intensity in the scene by setting the corresponding uniform variable. The integration of various uniform variables like enableLighting (for toggling lighting calculations) and ambient (for ambient light intensity) plays a crucial role in the visual enhancement of the rendered scene through lighting algorithms.

6. MeshFS Variable

- **Condition Check:**

if(showTex && enableLighting): This conditional statement checks if both showTex and enableLighting are true. These two boolean uniforms likely control whether to display a texture (showTex) and whether to apply lighting calculations (enableLighting).

- **Lighting Calculations:**

vec3 normal = normalize(v_normal);: Normalizes the interpolated surface normal vector (v_normal) to ensure it has a length of 1. This step is crucial for accurate lighting calculations.

vec3 lightDir = normalize(lightPos);: Computes a normalized vector representing the direction from the fragment being shaded to the light source. This assumes that lightPos contains the position of the light source.

float diffuse = max(dot(normal, lightDir), 0.5);: Computes the diffuse lighting component. It calculates the dot product between the normalized surface normal and the normalized light direction. The max function ensures that the value doesn't fall below 0.5, effectively providing a minimum threshold for the diffuse lighting intensity.

vec4 finalColor = texture2D(tex, v_texCoord);: Retrieves the color from the texture at the current texture coordinate (v_texCoord).

gl_FragColor = finalColor * (diffuse + ambient);: Combines the texture color with the calculated lighting. The final fragment color is the product of the texture color and the sum of the diffuse lighting and an ambient lighting term (ambient). This effectively modulates the texture color based on lighting conditions.

This block of code integrates lighting calculations with texture sampling when both showTex (texture display) and enableLighting (lighting calculation) flags are true. It computes the lighting effect based on the surface normal, light direction, and texture color, resulting in a final fragment color that incorporates lighting information for a more realistic appearance.

7. UpdateLightPos() Function

- **Control of Light Position:**

The function utilizes keyboard inputs to control the movement of the light source in 3D space.

Arrow keys (ArrowUp, ArrowDown, ArrowRight, ArrowLeft) adjust the position of the light source along the Y and X axes based on the defined translationSpeed.

- **Light Position Update:**

const lightZ = -5;: Presumably, the light's position along the Z-axis is set at a fixed value of -5 units.

const lightPos = [lightX, lightY, lightZ];: Constructs an array (lightPos) representing the updated 3D position of the light source based on the changed X and Y coordinates and the fixed Z value.

- **Uniform Update (gl.uniform3fv):**

gl.uniform3fv(lightP, lightPos);: Updates the value of the uniform variable lightP (which holds the location of the light position uniform in the shader program) with the updated lightPos array. This function call uses gl.uniform3fv to set a 3-component vector (in this case, the light's position) as a uniform variable in the shader program. This function specifically sets a 3-element float vector into the uniform location lightP.

- **Error Handling:**

If lightP is null (meaning the uniform location retrieval failed), an error message is logged to the console, indicating the failure to locate the uniform responsible for holding the light position.

- **Logging:**

console.log(lightPos);: Outputs the updated light position vector to the console, allowing for a visual check of the altered position.

Overall, this function allows dynamic movement of the light source in the scene using keyboard inputs and ensures synchronization between the JavaScript logic and the WebGL shader program by updating the respective uniform variable with the updated light position.

CONCLUSION

In my WebGL exploration thorughout this project, I've learned how JavaScript influences shader programs, using uniform variables to manage lighting and material properties dynamically. Functions like enableLighting and setAmbientLight control lighting effects, while buffer operations handle essential data for accurate 3D rendering. WebGL's ability to execute complex tasks and its flexibility in real-time visual control are remarkable. The seamless JavaScript-graphics fusion creates interactive 3D web content, emphasizing the importance of error handling for seamless integration. This journey has emphasized continual learning in mastering WebGL and its vast potential in web-based graphics.