CSCI 522 – Game Engine - Fall 2024 – M3

USC ID: 1154164561
NAME: Srija Madarapu
EMAIL: madarapu@usc.edu

Link to the video:

https://drive.google.com/drive/folders/1yM5kecOPfxU5k8f84o1ZXIpgKC-ZRw4Z?usp=sharing

M3: Creating water effect. Based on the key pressed the speed(1 for high & 2 for low speed) or the wave distance(3 for high & 4 for low distance) changes

1. Model Preparation in Blender

- Build the Model: You create the plane model in Blender and export it as an FBX file using the command:
  ./fbx.sh ../AssetsIn/Maya/plane/plane.fbx mesh -p PlaneTest -flip z This flips the Z-axis so that the plane faces upward correctly in the game engine scene.

2. FBX Export Process

- Exporting from Blender:
  The FBX file generated contains information about the geometry, materials, and textures of the model. The command flips the Z-axis to ensure that the game engine's left-handed coordinate system is correctly applied.

- Checking Exported Dictionary:
  The PlaneTest dictionary contains multiple dictionaries such as IndexBuffers, normalBuffers, Meshes, Textures (empty), and others. These are used for rendering the model in the game engine.

3. Importing into Maya and Converting

- Import the FBX to Maya:
  After exporting the FBX from Blender, you import it into Maya. In Maya, the goal is to convert the FBX file into a .mb (Maya Binary) file for use in the game engine.

- Metadata Scripts:
  You create a Python script, WaterPlane.py, which contains the line t["mayaRep"] = "Maya/plane/waterPlane.mb". This allows the game engine to load the correct model from the .mb file in the Maya directory.

- Adjustments in Maya:
  You modify the object in Maya to ensure its scale, position, and rotation are correct. Freeze transformations to reset rotation and scale values without altering the object's orientation. Change working units to meters for consistency across tools.

- Export Level from Maya:
  Once the model is set in Maya, the scene is exported to a level file, including metadata and transformations, and then saved.

4. Texture Conversion

To ensure the correct format for the game engine, you convert textures from PNG to DDS using the texconv tool. The format used for the DDS file is B8G8R8X8_UNORM, as it's supported by the game engine. The command you use is:

```
texconv -f B8G8R8X8_UNORM -m 0 -y -o C:\projects\PEWorkspace\AssetsOut\PlaneTest\Textures C:\projects\M3\Image2.png
```

5. Game Engine Integration

- Maya Scene to Game Engine:
  After exporting the .mb file, the game engine loads the model from the scene using the metadata provided in the Python scripts and the textures.
- Texture and Material Setup:
  You ensure the materials are correctly linked to the textures by modifying the .lua files to reference the correct DDS files.
- Mesh and Materials:
  The game engine uses the mesh information from the .mesha files (found in Meshes) and material data from .lua and .meta files. These files store the vertex data, including normals, UVs, and positions, which are used in the shader program to render the model.

6. Coordinate System Issues

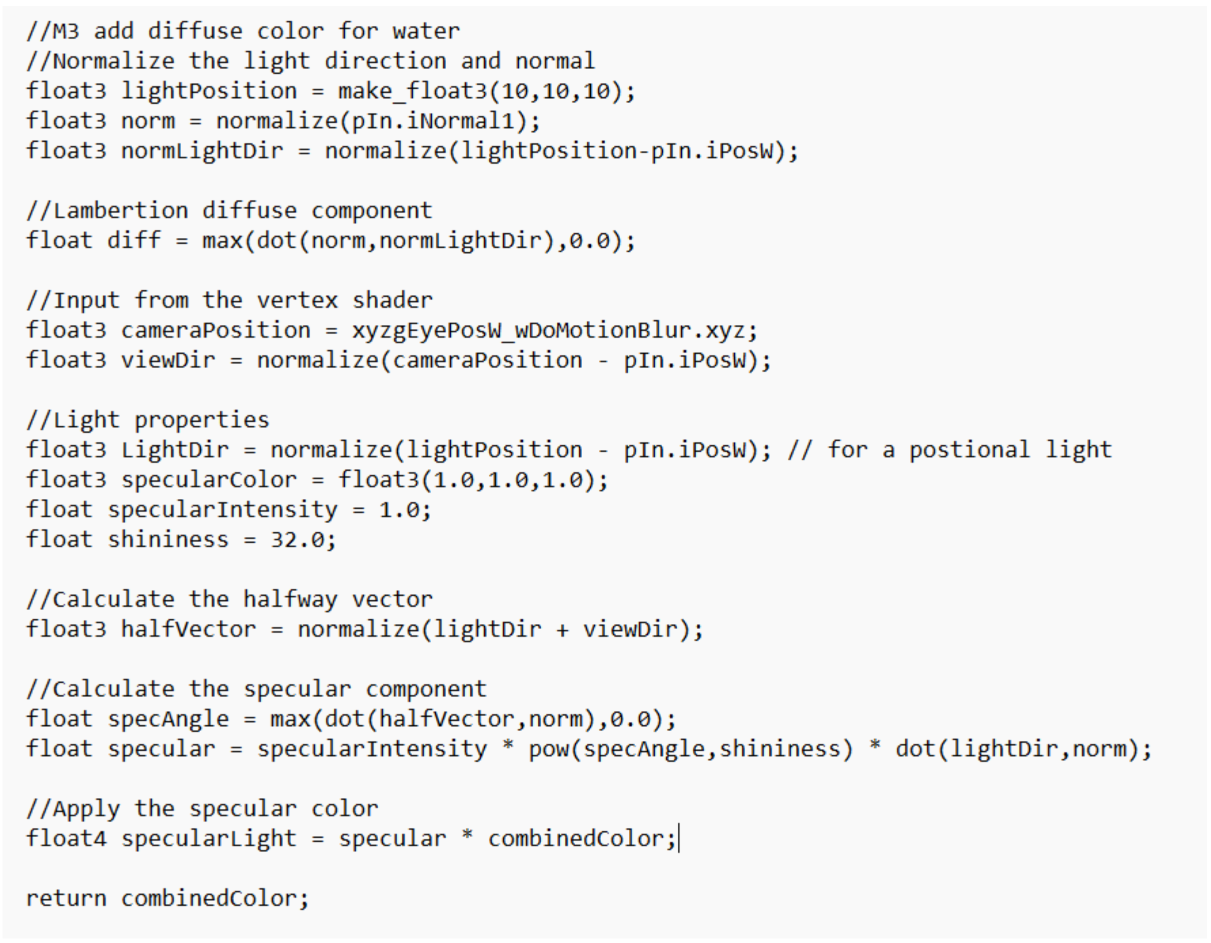
- Fixing the Orientation Problem:
  When you import the model into the game engine, the coordinate system seems incorrect (Y and Z are flipped). The issue arises from the Blender and Maya coordinate systems not aligning with the game engine's left-handed coordinate system.
  The Z-axis is flipped when exporting from Blender, but in the game engine, the Y-axis should represent up. The issue is solved by modifying the FBX export settings, as detailed by your TA.
- Scale and Rotation Issue:
  The scaling issue (where the model's scale is set incorrectly to 100) is fixed by adjusting the transformation values in Maya. After adjusting the rotation and scaling in Maya, you export the model again, ensuring that the transform matrix (m_base) is normalized and correct.

7. Mesh Handling in Game Engine

- Mesh Handling via Shader:
  The mesh is rendered in the game engine using the vertex shader. The material is assigned based on the mesh data in the .mesha file. The shaders (such as StdMesh_Shadowed_VS.cgvs) handle the color and material properties like diffuse and specular reflections, along with normal maps, bump maps, and other textures.

```
float wavePhase1 = dot(normDir3,pos.xy)/waveLength1 + time * waveSpeed1;
float waveHeight1 = sin(wavePhase1) * waveAmplitude1;
float wavePhase2 = dot(normDir3,pos.yz)/waveLength2 + time * waveSpeed2;
float waveHeight2 = sin(wavePhase2) * waveAmplitude1;
float wavePhase3 = dot(normDir3,pos.xz)/waveLength3 + time * waveSpeed3;
float waveHeight3 = sin(wavePhase3) * waveAmplitude3;
float wavePhase4 = pos.x/waveLength1 + time * waveSpeed1;
float waveHeight4 = sin(wavePhase4) * waveAmplitude1;

//Derivation with respect to x and z
float dWaveHeight1_dx = cos(wavePhase1) * waveAmplitude1 * (normDir1.x / waveLength1);
float dWaveHeight1_dx = cos(wavePhase1) * waveAmplitude1 * (normDir1.y / waveLength1);
float dWaveHeight2_dx = cos(wavePhase2) * waveAmplitude2 * (normDir2.x / waveLength2);
float dWaveHeight2_dx = cos(wavePhase2) * waveAmplitude2 * (normDir2.y / waveLength2);
float dWaveHeight3_dx = cos(wavePhase3) * waveAmplitude3 * (normDir3.x / waveLength3);
float dWaveHeight4_dx = cos(wavePhase3) * waveAmplitude3 * (normDir3.y / waveLength3);

//Calculate the tangent (T) and binormal (B) vectors
float3 tangent = float3(1,dWaveHeight1_dx+dWaveHeight2_dx+dWaveHeight3_dx,0);
float3 binormal = float3(0,dWaveHeight1_dz+dWaveHeight2_dz+dWaveHeight3_dx,1);

//Normalize the vectors
tangent = normalize(tagent);
binormal = normalize(binormal);

//Calculate the normal using the cross product
float3 normal = cross(binormal,tangent);
pos.y = waveHeight1+waveHeight2+waveHeight3;

//Calculate the partial derivtive of waveHeight4 with respect to x
float dWaveHeight4_dx = cos(wavePhase4) * waveAmplitude1 / waveLength1;
//Compute the adjusted normal
float3 adjustedNormal = normalize(make_float3(-dWaveHeight4_dx,1.0,0.0));
```

- Filtering in Shader:
  You filter material properties per mesh (e.g., diffuse color for the dress), either by changing material properties in the shader or filtering based on color in the vertex shader. Alternatively, a constant per material could be used, though it's more complex.



```
//M3 add diffuse color for water
//Normalize the light direction and normal
float3 lightPosition = make_float3(10,10,10);
float3 norm = normalize(pIn.iNormal1);
float3 normLightDir = normalize(lightPosition-pIn.iPosW);

//Lambertion diffuse component
float diff = max(dot(norm,normLightDir),0.0);

//Input from the vertex shader
float3 cameraPosition = xyzgEyePosW_wDoMotionBlur.xyz;
float3 viewDir = normalize(cameraPosition - pIn.iPosW);

//Light properties
float3 LightDir = normalize(lightPosition - pIn.iPosW); // for a postional light
float3 specularColor = float3(1.0,1.0,1.0);
float specularIntensity = 1.0;
float shininess = 32.0;

//Calculate the halfway vector
float3 halfVector = normalize(lightDir + viewDir);

//Calculate the specular component
float specAngle = max(dot(halfVector,norm),0.0);
float specular = specularIntensity * pow(specAngle,shininess) * dot(lightDir,norm);

//Apply the specular color
float4 specularLight = specular * combinedColor;

return combinedColor;
```

- **Sine Function**: The sin function models the oscillation of waves over time and space, with the wave height being proportional to the sine of the phase (wavePhase).
- **Wave Phase**: The wave phase (wavePhase1, wavePhase2, etc.) is a combination of position and time, which determines the oscillation of the wave. The wavelength (waveLength) affects how fast the wave oscillates in space.
- **Derivative**: The derivative of the wave height (using cos) gives the slope of the wave in the x and z directions, which is crucial for computing the surface normal.

  So, the **sine function** is used to generate the **wavelength** (through its phase argument), controlling the oscillation of the wave over both space and time.

8. Water Wave Simulation

- Adding Waves:
  To simulate water waves, you combine sine functions together and pass the normal vectors calculated using partial derivatives to the pixel shader for rendering water effects.
- Adding Speed and Wavelength Control:
  You add functionality to control the speed and wavelength of the water waves using game events. This is done by:
  1. Adding EVENT_WATER_SPEED and EVENT_WATER_WAVELENGTH to handle keyboard input events (e.g., KEY_1/2/3/4_HELD).
  2. Pushing these events to a queue and processing them in the GameThreadJob.

3. Passing the values for speed and wavelength to the RootSceneNode, which will affect the water's rendering in the shader.