Structure:

Explain the Structure and Goals of the Project

Structure

1. **Data Loading**
    - Read a large `.pts` file containing coordinates (x,y,z)(x,y,z)(x,y,z), colors (r,g,b)(r,g,b)(r,g,b), and normals (nx,ny,nz)(nx,ny,nz)(nx,ny,nz).
    - Store this data in GPU buffers (VBO) for efficient rendering.
2. **Shader Program**
    - Vertex Shader: Transforms each point's position into clip space, passes along color and normal.
    - Fragment Shader: Implements simple Lambertian lighting using the normal, light position, and color. Also discards fragments outside a circular region for round point sprites.
3. **Rendering Loop**
    - Clears the screen each frame, sets uniforms (point size, light position, etc.), and draws the entire point cloud via `glDrawArrays(GL_POINTS, ...)`.
    - Allows camera movement for interactive visualization (e.g., orbit, pan, zoom).

Goals

- visualize a high-density point cloud (e.g., a dinosaur model) in real time.
- demonstrate basic lighting (single point light) and color usage from the dataset
- enable interactive exploration (camera movement, adjustable point size, etc.).
- Learn How OpenGl works
- learn to use glfw window creation
- learn how to make callbacks and how to manipulate the window size, camera and point size interactively
- learn how to program shaders and how to make a basic shader class
- use and understand the Render Pipeline from start to finish
- understand how a basic camera works
- try to handle large point clouds somewhat efficiently
- learn how to display and light point clouds
- learn how to render the points

- learn how the different libraries work together

Explanation of the Approach & Libraries Used

Short goal formulation:
We want to build an OpenGL-based viewer for a 3D point cloud stored in a `.pts` file (with positions, colors, and normals), and render it using GPU point rendering, with interactive camera controls.

First we make use of what we already have provided in the model_loading.cpp of the LearnOpenGL github repo. This means we include the following libraries and classes:

1. **GLFW**
    - Handles window creation, input (keyboard/mouse), and the main event loop.
2. **GLAD**
    - Manages OpenGL function pointers, ensuring the correct OpenGL API calls are available. Since OpenGL functions are provided by the GPU driver and not directly through headers, GLAD dynamically loads them at runtime.

3. **LearnOpenGL Framework**
    - We include the Camera and Shader Classes we learned about in the course. The Camera class enables us to move and look around and zoom. The Shader Class is a wrapper for handling OpenGL shaders. It makes the code more readable, maintainable and is more convenient. It has built in error checking and is used to log errors, load shaders, compile and link, bind uniforms, activate the shaders and much more.
    - OpenGL itself is considered an API, but it's rather a specification. It specifies what the result of each function should be and how it should perform it. The implementation is up to the developer of the drivers.

Then we define a function to load the point cloud. We inspect the file and see that it contains the **(x, y, z)** coordinates, **(r, g, b)** colors, and **(nx, ny, nz)** normals for each point. We removed the comment that indicates this, which was not necessary and we could have just skipped that line, but it ended up working that way.

We go on to define callback functions for the GLFWwindow to resize the window and to track mouse and keyboard input. Furthermore, we set the screen size, initialize the camera with a position, define variables to track the mouse position and define some timing variables so our updates and movement is frame independent.

Now we enter the main function, where we start off by initializing GLFW and creating the window with GLFW. Then we set the current thread to use this OpenGL context (also referred to as state), which is required before calling any OpenGL functions. We then register the callback functions to our GLFW window. The functions are defined after the main

function at the end of the file. We also load all the actual OpenGL functions from the driver and enable depth testing, so we make sure that closer objects visually occlude farther ones and don't end up with a flat image. Further, we load and compile the vertex and fragment shader and load the point cloud data. We then calculate the center of our model, which we planned to use for our camera position, but ended up not using it, so it is basically an artifact. We continue by generating a vertex array and a vertex buffer object, which both belong to the GPU side of OpenGL (live on the GPU VRAM). The vertex array object makes sure that we always know how to interpret the vertex buffer object, which holds the data for us. We then bind them, since OpenGL is a state machine. We do not pass data directly to glDrawArrays, but set up the context (state) and then OpenGL uses this context. After that, we copy the data to the GPU memory with glBufferData and make sure that we always point to the correct data inside our buffer. We do this by specifying stride length, positions, and more, which gets stored in the currently bound VAO, so we correctly point at the colours, positions and normals in our buffer. After that we unbind the VAO and VBO to not accidentally change things. We also enable that we can control the size of the points dynamically.

We move on to the rendering loop, which is a while loop that runs around 60 times per second and does, in short:

1. Time tracking (for smooth animation/movement)

2. Input handling (WASD, arrow keys, mouse) for camera movement and point resizing

3. Uniform updates (camera, light, etc.)

4. Rendering (clearing, drawing your point cloud)

5. Buffer swapping & event polling

Let's go into more detail for the points 3-5. Every rendering loop we have to clear the screen by emptying the color and depth buffers. Then we call a function to set the shader before setting uniforms. A uniform is a global shader variable, which is read only inside shaders and constant across a single draw call for all vertices or fragments. It is called uniform, because it is uniform across all processed elements (every vertex or fragment in a single draw call will see the same value). They do not change per draw call per vertex or fragment. They handle stuff like camera, light position and color. They can be different per draw call and per object. We specify the point size, lighting position, lighting colour and that our points should be round. Then we set transformation uniforms. Ultimately we draw the points by binding the VAO and specifying the size and that it should be points, no triangles or lines. When we call glDrawArray(), a lot happens. Let's break it down:

1. VAO & VBO Configuration Is Used:

OpenGL looks at the currently bound VAO, which remembers which VBO to use and how.

2. Vertex Shader Runs ONCE per Vertex

Every point (pointdata) from the VBO is fed into the vertex shader. The vertex shader runs once per point. The output is a 2D screen-space position and other data like color and normal. At the beginning of the vertex shader we specify the version, what we get and from where and what we output. We are getting a position, color and normal vector and we output to the fragment shader a position (worldposition) for lighting, the normals for lighting calculation and the colour. The vertex shader also makes use of the model, view, projection matrices and the point size, which we all specified in the .cpp file. The main function of the vertex shader first converts the 3 component position into a 4D vector, because affine transformations (like translation, rotation, and scaling) can only be represented together using 4×4 matrices — and those matrices require 4D vectors to work properly. The fourth entry is denoted as w and set to 1.0 and gets a different value for each position vector (each vertex) after the matrix multiplications (especially projection matrix). We get the world position (we place the object in the world) by multiplying the 4D vector with the model matrix. We use this world position in the fragment shader for lighting calculations. We then multiply the world position with the view and afterwards the projection matrix. The view matrix changes the coordinate system to have the camera at the origin (camera/view space). After that, we apply the projection matrix, which converts the 3D coordinates in camera space to clip space.Further, we transform the normals differently because if we would scale them the same way, then they would not remain normals. We also pass on the colour the fragment shader and set the point size (screen size in pixels).

We now enter the rasterization flow. We collect our input and form primitives (points, lines or triangles), we specified points, from the vertex shader as input. We then check what primitives lie partially or fully outside of the view frustum (the visible camera box) and clip the ones that are outside and pass on the visible ones. Vertices outside of values from -w to w are clipped. We continue with the perspective division, which is automatically done by the GPU and transforms the 4D coordinates of the clip space into the Normalized Device Coordinates, by dividing each entry by a value that depends on the distance of the point to the camera. If the fourth component is large, then dividing by it leads to a smaller overall appearance. A short example: Imagine a square with 4 vertices (the corners) in 3D space. Now divide all the components by a large positive real number *w*. This leads to each vector being closer to each other than they were before (“closer” meaning in 3D space measured by whatever norm, they are all equivalent in finite space, but let’s say we use the Euclidean Norm). Similarly, dividing by a smaller *w* leads to them being farther apart and therefore appearing bigger. After this, we multiply the normalized coordinates to match our screen space (viewport transform). So each coordinate now represents a pixel on the screen, so the GPU knows exactly where to draw on the screen. Now each of the points we draw becomes a square sprite, more on that in the next paragraph. It could be that points overlap or that the point sprites are bigger than the one pixel, so we need to decide which point to draw. These sometimes overlapping (more fragments generated for same pixel location) potential pixels are called fragments. We call this breaking down of primitives into fragments rasterization.

For each fragment the GPU runs the fragment shader, which gets the lighting position, lighting color and the option to draw round points as uniforms and the normals, the world position and the color of the vertices as input, passed on from the vertex shader. Inside the fragment shader’s main function we first change the points appearance. OpenGL actually does not draw points, even if we specify it should draw points, but instead draws points as small square-shaped sprites, not literal single pixels – unless we keep their size at 1. To get

a round appearance, we can make use of the variable gl_PointCoord, which we have access to now since we specified that we want to render points. This variable gives us the coordinates of the current fragment position in the square sprite. Fragments are potential pixels. The square sprite ranges from (0,0) bottom left to (1,1) top right. What we do is move the center which is currently at (0.5,0.5) to (0,0) by subtracting -0.5 from both components. Then we check if the vector has a norm (Euclidean) greater than 0.5, if it does, then it is outside a circle and part of a corner which we want to get rid of. We write 'discard' in the fragment shader, to make sure this pixel is not rendered. Discard terminates the current fragment shader execution, which means its not written to the framebuffer. Basically, it is never drawn. We go on to calculate the Lambert (diffuse) lighting. We do no other lighting calculations since it was not specified in the task. We start off by normalizing both the normal vector and lighting direction, which is the lighting position minus the vertexe's world position. We do this, because the Lambert's cosine law needs normalized vectors. This law is simply taking the maximum of either 0 or the dot product of the normalized normal and normalized lighting direction. This dot product is 0, if the vectors are perpendicular, that is they get not lit or hardly (if we have ambient lighting) lit at all, and 1 (since we normalized them) if they are parallel, that is in this case they are either the same or point into the exact opposite direction. They cannot be parallel otherwise, since we normalized them and therefore a scalar multiple that is not equal to -1 or 1 of a vector (which is the definition of a parallel vector) would not be normalized anymore. We do not allow negative values, since in that case the lighting would be behind the surface of our object, which would not be a physically accurate representation of how real light works. Anyway, we end by specifying the litColor equal to the Lambert or diffuse value we just calculated times the color of the light times the original color. We output only this color with a fourth component which is the alpha value, which stands for transparency. It can be between 0 (fully transparent) and 1 (not transparent at all).

This color is written to the framebuffer, which is a chunk of memory where OpenGL draws the final image before it gets shown on the screen. It is like a canvas where each pixel gets painted by the shaders. This stage can contain different buffers, like depth and stencil buffers. Depth buffers keep track of how far each fragment is from the camera to ensure that closer objects get drawn in front of farther ones. Stencil buffers tell OpenGL where we are allowed to draw and where not, for example for creating mirrors or outlines and so on. Here gets decided which fragment actually turns into a pixel, via alpha testing, stencil and depth testing. Important to note is that OpenGL uses double buffering to prevent flickering. we have the front buffer, which is currently shown on the screen and the back buffer, what is currently getting drawn.

We go back to the .cpp file after the glDrawArrays call. Here we call glfSwapBuffers, which changes our current image to the one we just finished drawing and then we poll the events, which process all pending input and window events.

This concludes the rendering pipeline and render loop and our approach.

Explore the Dataset & Weakness of the Visualization

Dataset

- A large `.pts` file with millions of points, each having position, color, and normal data.
- Represents a scanned dinosaur skeleton.

Weaknesses of the Current Visualization

1. **Performance on Very Large Datasets**
    - Rendering millions of points in a single pass can cause performance drops on less powerful GPUs.
    - No Level of Detail (LOD) or out-of-core rendering is currently implemented.
2. **Lack of Advanced Shading**
    - Only simple diffuse lighting is used; no shadows or advanced shading models. No ambient or occlusion light is built in.
3. **Clutter & Occlusion**
    - Points can overlap visually, making it hard to see internal or hidden geometry. Especially if point size is increased.
    - No transparency or specialized “splatted” blending is applied.
4. **Sparse and wrong lit appearance of the model**
    - We have holes and fragmented silhouettes due to limited point samples.
    - Depending on the view angle we can see large holes and empty regions. Angles that are not responding to scanning direction.
    - Due to this, points that are rendered in the back shine through to the front.
    - Real-world scans have noise, duplicates, missing points.
    - This can cause rendering glitches or poor lighting.
    - Computing normals on points requires additional methods like Principal Component Analysis, which estimates normals. Depending on the density of points this leads to good or bad results, which in turn the lighting depends on.

Challenges When Visualizing Point Cloud Data

1. **Data Size & Memory**
    - High-resolution scans reach hundreds of millions of points, requiring large amounts of GPU memory.
    - Even transferring the data to the GPU can be time-consuming. We have to make efficient use of draw calls.
    - This could lead to low FPS, long loading times and out of memory.

2. **Noise & Incomplete Surfaces**
    - The scan has holes, occlusions, and noisy points. Sometimes points in the back got rendered to the front.
    - Densely packed points do overlap and introduce noise and reduce visual clarity.
3. **Point-Based Rendering Limitations**
    - Points have no explicit connectivity. There are often large gaps and sparse areas that look incomplete.
    - Lighting can be tricky if normals are inaccurate or if the dataset is inconsistent, which I do not think is the case here.
4. **Performance vs. Quality**
    - Larger points can mask holes but may create blobby surfaces.
    - Smaller points show detail but require more fill-rate and might appear patchy.

How Could the Visualization Be Improved?

1. **Level of Detail (LOD)**
    - Implement LOD techniques (e.g., octrees, hierarchical clustering) to reduce the number of points drawn at once.
    - Dynamically adjust point density based on the camera distance.
2. **Advanced Shading & Shadows**
    - Use more advanced lighting models (Phong, PBR) or **screen-space ambient occlusion** for more realistic depth cues.
    - Implement **shadow casting** if performance allows (e.g., shadow maps from the point light).
3. **Splatting & Blending**
    - Instead of simple circular sprites, use elliptical weighted average (EWA) splatting or Gaussian splats for smoother surfaces and fewer gaps.
    - Blend overlapping points to reduce harsh edges.
4. **Data Filtering & Noise Reduction**

- Apply pre-processing to remove outliers and smooth the normals.
- Fill small holes or combine close points for a cleaner visual.

5. **User Interaction & Controls**
    - Add keyboard shortcuts for toggling advanced features (e.g., round points on/off).
    - Implement an on-screen UI to select color maps, lighting modes, or LOD thresholds.
6. **Gathering more points to make a fuller appearance**
7. **Use surface reconstruction to turn points into meshes and therefore have a proper surface**
8. **Add further lighting.**