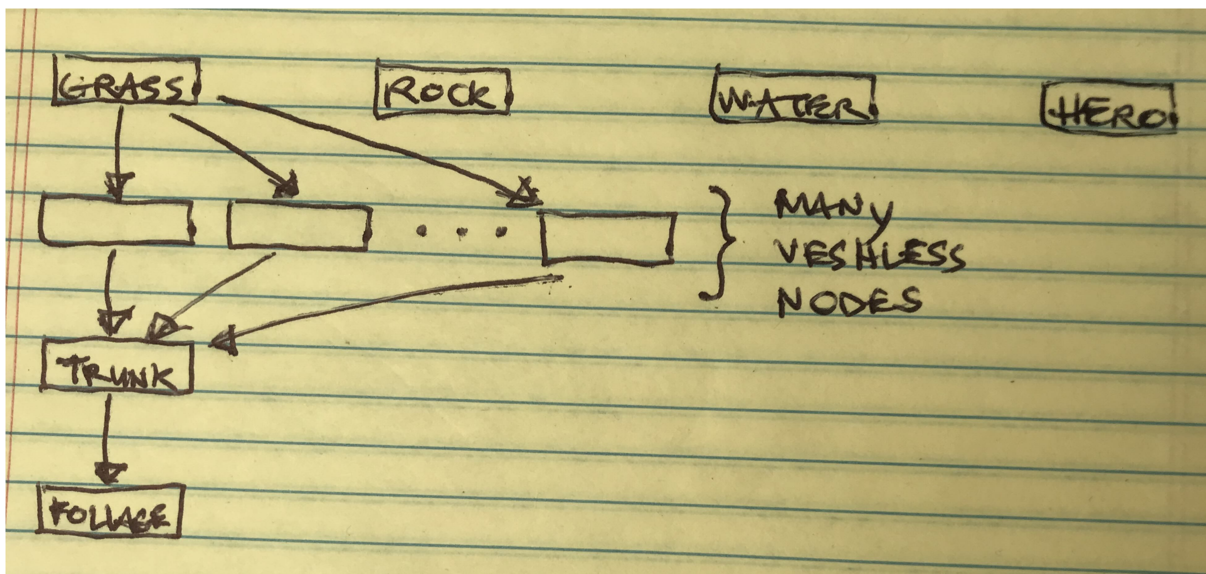
A.A. Clipping at the near plane is necessary, because without it two bad things can happen. First, vertices at the camera plane can cause division-by-zero errors when we do homogeneous division. (And therefore vertices near the camera plane can cause numerical instability.) Second, objects behind the camera can appear in the scene, although they should not.

A.B. Clipping at the far plane is not necessary for either of the two reasons in A.A, or for any other reason. In fact, if realism is our primary concern, then clipping at the far plane is undesirable, because it causes the scene to be artificially "cut off" at a certain distance in front of the camera. But people do it anyway, because it improves rendering speed, because it prunes parts of the scene. The harm to the realism is mitigated by artistic design, such as atmospheric effects or simply always having objects between the camera and the far plane.

B.A. The swap chain is the sequence of raster images, that are shown to the user. For example, in a double-buffered application, the swap chain has length two. While we are showing one frame to the user, we can be working on constructing the next frame. The swap chain involves a lot of Vulkan machinery, including images (and hence buffers) and image views.

B.B. We rebuild the swap chain whenever the window is resized. The simple reason is that all of the swap chain elements have the same size as the window, and so they need to be laid out anew in memory. The more subtle reason is that larger windows require more memory, which could (at least in principle) affect the length of swap chain supported. We also have to rebuild the pipeline, because that depends on the swap chain.

C.A. A reasonable answer is shown below. You may have different details at the bottom. The most important thing is that there are many vesh-less nodes that all point to the same tree.



C.B. Processor time? Linear. The command buffer holds a rendering command for each path through the scene graph from the (eldest) root to the leaf.

Memory for meshes? Constant. All of the trees use the same two (or more) meshes.

Memory for textures? Constant. All of the trees use the same two textures.

Memory for scene-wide uniforms? Constant. It has nothing to do with the trees.

Memory for body-specific uniforms? Linear. A separate copy is needed for every path through the scene graph from the root to the leaf, because these paths can produce different modeling isometries.

D.A. The diffuse term is

$$i_{\text{diff}} * \vec{c}_{\text{diff}} * \vec{c}_{\text{light}},$$

where i_{diff} is the diffuse intensity, $\vec{c}_{\text{diff}}$ is the diffuse surface color, and $\vec{c}_{\text{light}}$ is the incoming light color. The first asterisk is scalar multiplication, and the second asterisk is vector modulation.

D.B. The diffuse intensity is essentially $\vec{u}_{\text{normal}} \cdot \vec{u}_{\text{light}}$, where $\vec{u}_{\text{normal}}$ is the unit outward-pointing normal vector for the surface (typically interpolated from vertex attributes) and $\vec{u}_{\text{light}}$ is the unit vector pointing from the fragment toward the light source. Thus the diffuse intensity is close to 1 when the light hits the surface directly and close to 0 when the light glances off the surface. However, if the light source is behind the surface, then the dot product above is negative. We don't want that, because negative light might cancel out some of the other lighting calculations. So actually

$$i_{\text{diff}} = \max(0, \vec{u}_{\text{normal}} \cdot \vec{u}_{\text{light}}).$$

D.C. For a positional light, the direction $\vec{u}_{\text{light}}$ from the fragment to the light is calculated as follows. In the vertex shader, we output the world position $\vec{p}_{\text{vertex}}$ of the vertex. Between the vertex shader and the fragment shader, it gets interpolated to the world position $\vec{p}_{\text{fragment}}$ of the fragment. (This interpolation is perspective-corrected, so it behaves mathematically as if it happens in world space, which is good, although it literally happens in screen space.) The fragment shader is also given the world position $\vec{p}_{\text{light}}$ of the positional light, as a scene uniform. Then

$$\vec{u}_{\text{light}} = \text{normalize}(\vec{p}_{\text{light}} - \vec{p}_{\text{fragment}}).$$