A. [I'll omit the drawing. It should show an origin or some other indicator of location, because the distinction cannot be seen without one. In the first case, the face rotates about the origin and then translates away. In the second case, the face translates away and then revolves about the origin. The final orientation is the same in the two cases, but the final position is not.]

B. [A diagram would help, but I'll omit it.] The attributes arrive in local coordinates $\vec{\ell}$ relative to the origin of the mesh. They are homogenized by appending a 1 to them. The world coordinates $\vec{w} = M\vec{\ell}$, where M is the modeling transformation from the scene graph, express the location of the vertex in the virtual world. The eye coordinates $\vec{e} = C^{-1}\vec{w}$, where C is the camera transformation, express the location of the vertex relative to the camera. The clipping coordinates $\vec{c} = P\vec{e}$, where P is the projection, are used in the clipping algorithm. After clipping, the normalized device coordinates (although we have not used that term in class) $\vec{d} = \vec{c}/c_3$ express the vertex as a point in a standardized cube $[-1, 1] \times [-1, 1] \times [-1, 1] \times \{1\}$. Finally, the screen coordinates $\vec{s} = V\vec{d}$, where V is the viewport, express the vertex as a point on the screen, with depth information attached. These coordinates $\vec{s}$ are passed to `triRender`, along with any other attributes that have come along for the ride.

C. Yes, our 3D graphics engine is powerful enough to handle 2D graphics as a special case. Use a trivial camera rotation. Set up a trivial orthogonal projection as we discussed in class, with $\ell = b = 0$, $r = w - 1$, $t = h - 1$. If we use $n = -1$, $f = -3$, and a camera world position of $\langle 0, 0, 2\rangle$, then the world $z = 0$ plane runs through the middle of the viewing volume. So, even though clipping is still happening, it has no visible effect. In `transformVertex`, assign each 2D vertex a world z-coordinate of 0. That's it...mostly.

In theory, all rendered pixels will have equal depth values. In practice, the foregoing calculations might introduce some numerical noise into the depths, causing overlapping triangles to "fight" in the depth buffer, causing a visual artifact called "stitching". Here are two strategies for mitigating stitching. The first strategy is to add a flag to the triangle renderer, that lets us disable the depth test. That's cheating, because it requires editing triangle.c. I mention it only because it's what a real rendering system would do. The second strategy is to give the triangles slightly different world z-coordinates in `transformVertex`. That is, when rendering the first mesh, use z-coordinate 0. When rendering the second mesh, use $z = \epsilon$, where ϵ is a small number. When rendering the third mesh, use $z = 2\epsilon$. And so forth. The result is identical to the painter's algorithm, in that later rendering occludes earlier rendering.

D. Yes, it is possible to implement a scene graph without going into homogeneous coordinates. We use the homogeneous trick for speed. At depth d in a scene graph, `transformVertex` has time complexity $\Theta(1)$ with this trick and time complexity $\Theta(d)$ without it.

To compute the modeling transformation for a node at depth d, we have to perform a sequence of $d+1$ rotations alternating with $d+1$ translations. So the time cost of the modeling transformation on each vertex is proportional to d. By using homogeneous matrices, we are able to collapse this sequence of rotations and translations into a single matrix. That is, each node requires one matrix multiplication to make its modeling transformation (from its transformation and its parent's) and just one matrix multiplication per vertex.

E0. If we didn't clip at the near plane, then we might get division-by-zero errors in the homogeneous division. Even if we didn't get these fatal errors, we might divide by numbers close to zero, which might produce numerically unstable results. Also, we might render geometry behind the camera, producing other strange artifacts.

E1. We might clip at other planes for speed. For example, if a triangle's vertices' e_1-coordinates (the y-eye-coordinates) are too large, then the triangle will not be drawn, because it would have to be drawn above the window on the screen. So maybe we should detect this case and avoid all of the rasterization, interpolation, pixel shading, etc. for that triangle.

E2. [A picture helps, but I'll omit it.] I have an example where the intersection of a triangle with the viewing volume is a heptagon (seven-sided polygon). A heptagon can be partitioned into five triangles, but no fewer than that. So the clipping algorithm might emit as many as five triangles per incoming triangle. Is there a bigger example?