A. [Rather than draw the graph, I give the same information in text. By the way, if you examine the `#include` statements in 130mainDepth.c, you will see the same files in roughly opposite order. I do not mention main.c, because it is the "user" of our graphics engine, not part of it. I did not penalize students who included main.c in the graph. I also did not penalize students who omitted pixel.h.]

I would draw six of the nodes in this order, from top to bottom:

- mesh.c has outgoing arrows to triangle.c and shading.c because of its rendering algorithm. It has arrows to depth.c and texture.c, just because it must pass those data types to triangle.c. It should also have an arrow to vector.c because of the mesh builders.
- triangle.c has arrows to vector.c, matrix.c, shading.c, texture.c, depth.c, and pixel.h.
- shading.c has an arrow to texture.c.
- texture.c has an arrow to vector.c.
- matrix.c has an arrow to vector.c.
- vector.c has no outgoing arrows.

Then I would draw these nodes in a second column:

- depth.c has no outgoing arrows.
- pixel.h has no outgoing arrows.

B. Depth buffering solves the occlusion problem, which is: When multiple objects in a 3D scene are competing to be rendered in a given pixel, the one that is closest to the viewer should win.

The only other solution that we've discussed is the painter's algorithm, which is: Sort the scene objects from distal to proximal, and render them in that order.

We prefer depth buffering because the painter's algorithm handles only convex, non-intersecting scene objects. It does not handle non-convex objects and intersecting objects. Depth buffering handles all of these objects easily. The only drawback to depth buffering is the extra memory required to hold the depth buffer. Well, another drawback is occasional "stitching" or "Z-fighting" effects from objects at numerically indistinguishable depths.

C. The vertex shader has access to uniforms and attributes. If there are any texture coordinates among the uniforms and attributes, then they have not yet been interpolated. So the vertex shader could sample textures only at uniform or attribute texture coordinates — not varying. But texels sampled with uniform coordinates might as well be in the uniforms, and texels sampled

with attribute coordinates might as well be in the attributes. Therefore, giving the vertex shader access to texture sampling seems to improve its flexibility very little.

[This was the most speculative question on the exam. There is not really any right answer. I just wanted to see what students said. For example, a good thing to say was that texture coordinates haven't yet been interpolated.]

[Actually, attribute texture coordinate sampling could be useful. It would allow us to implement highly complicated transformations of the attributes, using the texture as a lookup table. For example, suppose we render mesh A with texture B, and in a different part of the scene we render mesh A with texture C. We could use the differing textures to implement complicated transformations of A's attributes.]

D. [Several students had trouble understanding this question, but it is not an unreasonable question. It was foreshadowed in our Day 06 homework, when we implemented 080mesh.c. I also alluded to it in the Day 10 study questions. Also, multiple students raised these issues with me in office hours, without any prompting from me.]

One algorithm for `meshRender` is:

1. For each triangle:
   (a) Transform the first vertex from attribute to varying using the vertex shader.
   (b) Transform the second vertex.
   (c) Transform the third vertex.
   (d) Pass the three varyings just made to `triRender`.

The problem with this algorithm is that, if a vertex is used in multiple triangles, then it is transformed multiple times by the vertex shader. So there is wasted work. [A similar waste issue motivated our mesh data structure in the first place.] So here is a second algorithm for `meshRender`:

1. Allocate a large array to hold the varying versions of all of the vertices.
2. For each vertex:
   (a) Transform the vertex from attribute to varying using the vertex shader.
   (b) Store the varying vector in the large array.
3. For each triangle:
   (a) Locate the triangle's three varyings in the large array.
   (b) Pass the three varyings to `triRender`.

The problem with this algorithm is that it uses a lot of extra memory. So what we have here is a classic time-space tradeoff. Which algorithm is better depends on our resource constraints — for example, how much memory we have.

[In today's computers, using memory can take a great deal of time, and hence the second algorithm may be slower than the first, even though it does fewer computations. It all depends on the relative speeds of the CPU and RAM, the sizes of the memory caches, the size of the mesh, etc. CS 311 students are not expected to know all of this, but students who have taken CS 208 might know something about it.]

[Several students instead discussed depth testing before or after the fragment shader. The former is faster and the latter more flexible. A good answer along those lines earned lots of points. However, this change requires semantic changes to `meshRender` or the shaders. It is not merely a semantically equivalent implementation change to `meshRender`. So it is not as good of an answer as the one given above.]

E.A. There are no technical restrictions on how attribute vectors are formatted. The attributes can be anything, and they can be listed in any order. As long as the vertex shader knows their format, it can "parse" them to produce varying vectors.

Here's another angle. In mesh.c, our 2D convenience builders produce attributes in the format XYST, and our 3D convenience builders produce attributes in the format XYZSTNOP. But you don't have to use these convenience builders. You can build any mesh you want using the other methods of `meshMesh`. And you can, for example, put X and Y in the 6th and 4th attribute components. Nothing in `meshRender`, `triRender`, etc. will break.

E.B. There is one crucial restriction on how varying vectors are formatted: The X and Y components must come first. That's because the rasterizer is hard-wired to rasterize over the first two components. As it rasterizes, it computes linear interpolations, again based on the first two components. If we put X and Y in the 6th and 4th varying components instead of the 0th and 1th, then `triRender` would break.

[There is no theoretical reason why we had to write `triRender` this way. You can imagine making the components-over-which-to-rasterize-and-interpolate customizable. The rendering algorithm would take, as part of its many inputs, the numbers that we've been calling `mainVARYX` and `mainVARYY`. Commonly they would be 0 and 1, but you could change them to 6 and 4 if you really wanted. There is no theoretical obstacle to doing this, but we have not done this, and we could not do it without editing triangle.c, in contravention of the problem.]

[Some students said that the varyings must begin with X, Y, and Z. But the fragment shader does not have to produce its D value from the third component of the varying vector. In fact, I've done demos in class where D comes from the uniforms.]