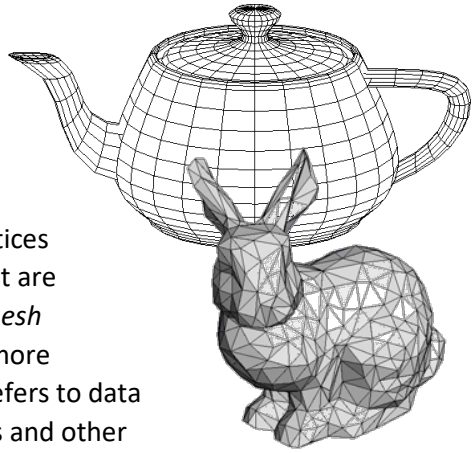
3D Models and Materials



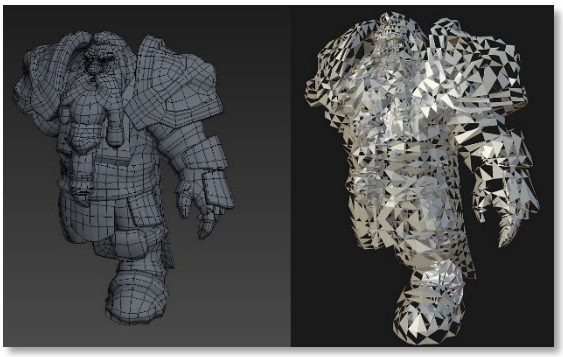
What are 3D Models?

3D models are, in their simplest form, sets of vertices. These vertices define one or more collections of polygons – often triangles – that are arranged to form three dimensional shapes. In addition to this *mesh* information, 3D models also potentially have data about one or more *materials* that should be applied to its polygons. Material here refers to data about how the mesh should look when rendered: colors, textures and other surface properties. Some 3D models are even made up of a hierarchy of multiple, distinct meshes, each with their own transformations and materials.

The terms *mesh* and *model* are sometimes used interchangeably, but for our purposes: a *mesh* is simply a set of vertices and a *model* is a set of one or more meshes, possibly with material information, often stored in an external file.

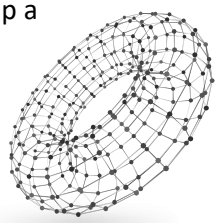
> *Our 3D models will be extremely simple to start with: a single mesh and no material information. We'll be defining our materials through code so they perfectly match our engine's capabilities.*

While many 3D modeling packages, like Maya and Blender, allow artists to define their models with quads (4-sided polygons), we need our meshes to be made up of exclusively triangles to actually render it. An artist could triangulate their model before exporting it, or we could triangulate it ourselves when loading. Either way, this is a step that must occur before a GPU can rasterize the mesh properly, or you end up with a result like the one shown to the right.



Vertex Data

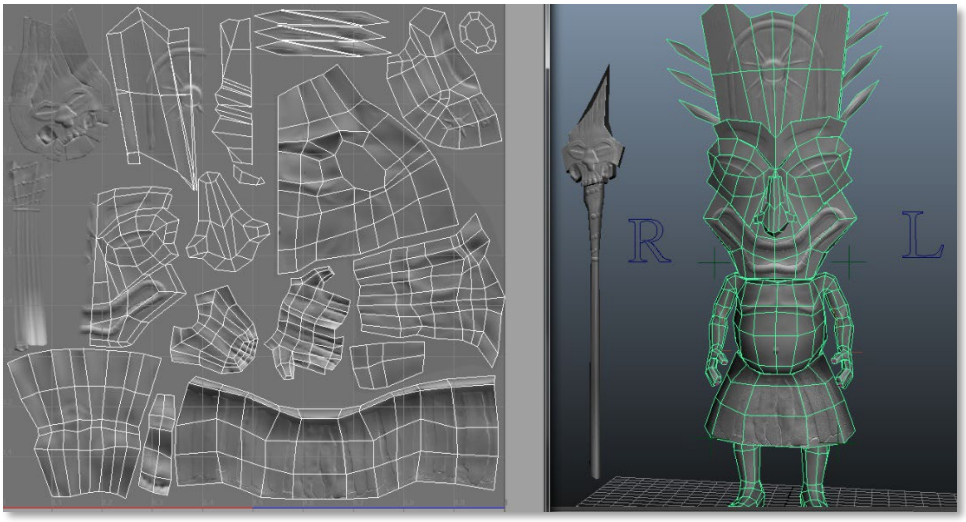
Vertices often store multiple pieces of data. At a minimum, the vertices that make up a 3D model need **positions** in 3D space. That is enough to define the overall *shape* of the model, but if we want our models to look more interesting than a solid color, we'll need information about the surface of the model. For our basic game engine, we'll need **texture coordinates**, also known as UV coordinates, and **surface normals**. Both of these are usually created at design time in whatever 3D modeling package the artist was using.



> *Notice that I haven't mentioned vertex color here. Once we start using a texture for the colors of our pixels, we don't really need each vertex to have its own color. It's not impossible, it's just not as useful anymore. Some advanced techniques might use vertex color data for other purposes, but we'll be removing it from our vertex definitions going forward.*

Texture (UV) Coordinates

Texture coordinates, or UV coordinates, define how a 2D texture (an image) is applied to a 3D model. In other words, how the image wraps around the model's triangles. Since this data is defined per-vertex, an artist can exactly control exactly which section of an image corresponds to each triangle.

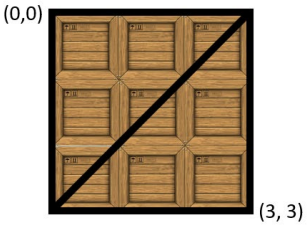


In the example above, the image on the left maps to the 3D model on the right through the use of texture coordinates. The artist decided to lay different parts of the model – the mask, the torso, the skirt, etc. – out in distinct areas and then paint the final texture onto those areas.

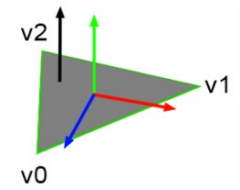
Texture coordinates are usually within the 0 – 1 range, where (0,0) is one corner of the image and (1,1) is the opposite corner. In this way, UV coordinates are essentially a percentage across the image, rather than precise pixel coordinates. This allows artists to disregard the final resolution of a texture, as the coordinates will map to an image of any size. Also note that in HLSL, texture coordinate (0,0) is the *top*-left corner of the image while in GLSL, (0,0) the *bottom*-left corner.

Why are texture coordinates often called UV coordinates? Texture coordinates are usually two dimensional, and the letters XYZW are already taken by our homogenous coordinates. The next two letters closest to the end of the alphabet are U and V.

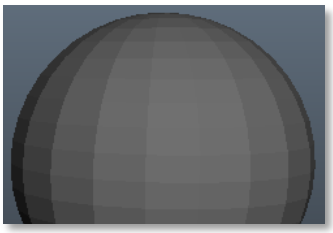
UV coordinates might also go outside the 0 – 1 range. The exact meaning of coordinates outside 0 – 1 depends on the application, but it is often used to repeat a texture across a surface. For instance, see the two sets of triangles below. On the right, the UVs go from 0 to 3, causing the texture to repeat three times across the surface. In essence, the 0-3 range really means "use the 0-1 range three times", which can be calculated as follows: $imageUV = vertexUV \ \% \ 1.0$



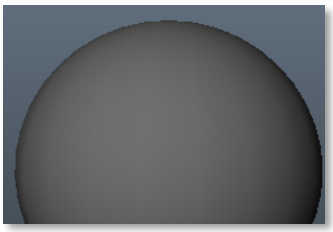
Surface Normals



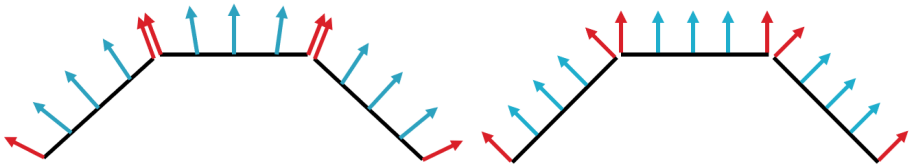
Not only do we need to know *where* the triangles are; we also need to know *which way they face* in 3D space. We could calculate this information ourselves given the three vertices of a triangle: the cross product of two edges of the triangle would give a vector perpendicular to the triangle. Unfortunately, this is *not* something we can do in a vertex shader, as we only have information about a *single vertex* at a time, not an entire triangle. So instead, this information is calculated ahead of time and stored at each vertex.



If each vertex of a single triangle had the same normal, every pixel of that triangle would "face" the same direction. After performing our lighting equations, the shading on each triangle would appear distinct from neighboring triangles, giving a faceted look. This is often referred to as "hard shading" or "flat shading", and is often employed when an artist is specifically going for a "low-poly" look.



If, however, each vertex of a single triangle had a *different* normal, then each pixel of that triangle will also have a slightly different normal after the values are interpolated by the rasterizer. This results in a different "direction" for each pixel, meaning the overall shading will appear smooth, hence the term "smooth shading".



Smooth (left) vs. Hard (right) shading

Whether a model is "smooth" or "hard" shaded is determined by the data in the model itself, and is not something we need to account for. If your model is defined the wrong way, get a different model!

Common 3D Model File Formats

If we want to load existing 3D models into our engines, we'll need to read them from files. There are numerous file formats for 3D models: some are text-based, some are binary, some are for specific programs, some are more universal, and so forth. We'll be using a very simple, text-based, universal 3D model file format called .OBJ, though other, more advanced formats could certainly be used.

.OBJ Format

The .OBJ, or Wavefront OBJ, model format can represent basic vertex data: *positions*, *texture coordinates*, *surface normals* and the arrangement of this data into *faces* (polygons). Sounds like exactly what we need! Since this is a text-based format, if we can open and read a file in C++, we can use .OBJ files for our engine.

Example .OBJ File

Below is text directly from an OBJ file of a quad (two triangles). This was created in Maya, exported to a file and then pasted into this document.

```
v -0.500000 -0.500000 0.000000
v  0.500000 -0.500000 0.000000
v -0.500000  0.500000 0.000000
v  0.500000  0.500000 0.000000

vt 0.000000 0.000000
vt 1.000000 0.000000
vt 0.000000 1.000000
vt 1.000000 1.000000

vn 0.000000 0.000000 1.000000
vn 0.000000 0.000000 1.000000
vn 0.000000 0.000000 1.000000
vn 0.000000 0.000000 1.000000

f 1/1/1 2/2/2 3/3/3
f 3/3/3 2/2/2 4/4/4
```

You can see several different types of data, each of which starts with a different identifier: *v* for vertex position, *vt* for vertex texture coordinate, *vn* for vertex normal and *f* for face. *Face* here essentially means a triangle, though a face could technically have more than 3 vertices. Each group of numbers (like 1/1/1 or 2/2/2) after *f* represents a single vertex. The numbers in a single group are 1-based indices into the other data in the file. For instance, 1/1/1 means the "first v / first vt / first vn".

> *There are other features of .OBJ files, such as external material definitions, groups and so forth, that we won't explicitly need and won't be covering here.*

Loading 3D Model Files

If you can read a text file in C++, you can load an .OBJ file. However, things get more complicated when we need to process the data: is the coordinate system assumed to be right-handed or left-handed? Are the faces triangles or quads? Our code may need to handle these situations.

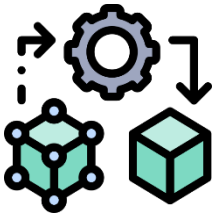
As such, I'll be providing very basic .OBJ file loading code for you in a future assignment. It won't support every possible .OBJ format feature, and it won't work for every .OBJ file you might find online, but it will be good enough to start with. If you're looking for a more sophisticated way of handling .OBJ files, I suggest looking into one of the following libraries:

- **TinyOBJLoader:** An easy to include and simple to use open-source OBJ model loading library.
- **The Open Asset Importer Library:** Open-source library to load most 3D model formats, including OBJ and FBX. This library supports a multitude of features above and beyond .OBJ files, but is much more complex to build and include in your own projects.

FBX Format

Another popular 3D model file format is FBX. While this is a proprietary format owned by Autodesk, there are multiple C++ helper libraries than can read and write FBX files. We won't be explicitly using FBX files, but I wanted to mention them as they are one of the most widely-used 3D model formats today. If you did want to dabble with this format, an official FBX SDK exists, though it is quite difficult to use. Instead, I highly suggest using the Open Asset Importer Library mentioned above.

Handling 3D Models in Code



Ok, we can open a file and read the contents. What do we do with it?

The first step is to update any and all parts of your code that define what a single vertex looks like. This means generally means updating any vertex-related structs in C++ and any code that uses or generates them. Likewise, any vertex-related structs in your HLSL shaders must also be updated to match. You most likely will not be using a per-vertex color anymore, so that data can be completely removed from any vertex structs. In its place, you'd need to add a 2-component vector for texture coordinates and a 3-component vector for the surface normal. Always ensure that your C++ and HLSL structs match in both size and order of variables!

Next, add the model-loading code to your Mesh class. Luckily, there isn't anything we substantially need to change about creating GPU buffers: the data you've read from the file – vertices and indices – can now be fed directly into the buffer-creation code you already have.

Lastly, you need to describe to the Graphics API exactly what a vertex looks like: how many pieces of data, their sizes and their purposes (semantics). This is done in Direct3D through an Input Layout. However, if you're using the SimpleShader helper library, an Input Layout will be generated for you from any vertex shader you load!

Materials

In the real world, "material" refers to the physical matter that makes up an object: wood, stone, cotton, iron, etc. Each of these materials has different physical properties that make them look distinct when light bounces off of them and hits our eyes.

In a game engine, our objects aren't truly "made" of anything. However, we want them to *appear* as if they are made of different materials when we render them. This is achieved through the use of specialized shaders to approximate light striking the objects, textures (images) to represent surface colors and other data that describes the properties of our virtual materials.

Material Data

What kind of data do we need to store to represent different materials? There is no universal list, as different engines using different lighting equations may require slightly different inputs. However, in general, some or all of the following data is usually part of a basic material system:

- Tint – A color tint to apply to the entire object
- Texture – An image representing the surface colors of the object
- Texture scale & offset – Scale and/or offset UV coordinates to adjust texture placement
- Roughness – How precise are reflections? (Sometimes called "shininess")
- Metal or non-metal? – Metals and non-metals inherently reflect light differently; which are we?
- Shaders – Which shaders are used to achieve this effect?

Some of this data might simply be a single number. Roughness, for instance, is just a value between 0 and 1. We could pass that single value to a shader and plug it in to all of the lighting equations for all pixels. Or, if we wanted different roughness values for each pixel of our object, this data could be stored in a texture of its own and read from that texture in a shader.

> *These kinds of decisions – which data to store and how to store it – need to be decided as we create a material system for our engines. As our engines become more complex, we'll need to revisit our material definitions to add new data and/or replace existing data to support more sophisticated techniques. As always, we'll start simple and expand.*

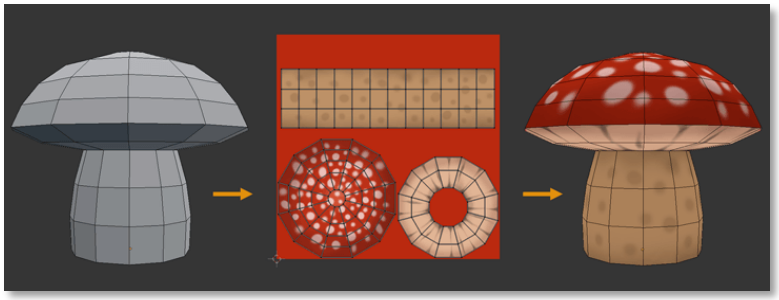
Tying Materials to Game Entities

Game entities are really just combinations of meshes, materials and transforms, so our entity representation now needs to track a material as well. Just as a single mesh might be used by more than one entity, a single material might be shared among multiple entities, too.

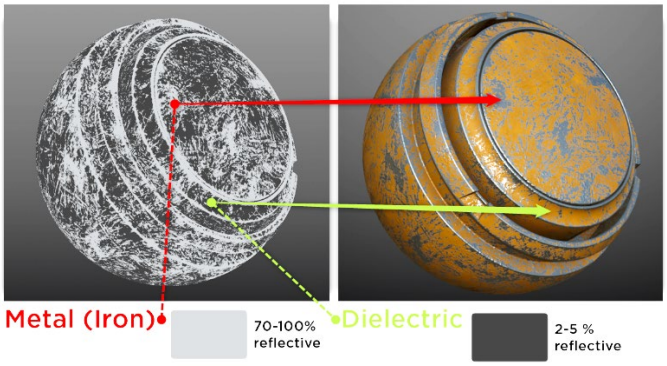
This means that each entity will have *exactly one* material associated with it. However, most 3D modeling packages allow artists to apply materials per-mesh *or per-triangle*, meaning a single mesh could contain multiple materials. And a single 3D model file could theoretically contain multiple distinct meshes, each with their own material. How do we handle all of that?

Unfortunately, multiple materials per entity doesn't map well to a real-time rendering engine. We're trying to process as many similar vertices and pixels as possible at once, which means we really want a single set of steps to follow for all of those pixels. Having completely different materials being applied to the same mesh is tricky. While it's not impossible, the overhead for it is a lot of work and often unnecessary for most 3D models.

However, we can still ensure a single mesh *appears* to have distinct materials by embedding much of the material data in textures to change the results per-pixel. As an extremely simple example, take the mushroom below. We want the three pieces – cap, gill (underside) and stalk – to all appear very different, even though it's a single mesh with a single material. To support this, the texture has distinct areas that give the *impression* of different materials, even though it's really just one.



In the more complex examples below, we have a single material that appears to be both metallic and dielectric (non-metallic) across its surface. Instead of having a single "metal" value for the whole entity, we use a texture with different values for each pixel. This is a common setup to allow visually distinct areas to exist within a single material. It does require the art assets, shaders and material systems to all be aligned in their inputs and representations of surface information.



How do engines like Unity handle models with multiple materials? By automatically splitting them into separate mesh/material combinations (entities) and linking them together using the transform hierarchy. This allows the end user to still treat it as a single game object while the engine treats it as multiple entities that each render separately!