**Model Import Procedure**

**FBX**

* + - Right handed, Y-up - no conversion needed
    - However, must be aware of I/O settings so you only pull the mesh and animation data
    - Mesh data contained within FbxMesh class, get with FbxNode.GetNodeAttribute
    - Transform for individual mesh contained in .::LclTranslation, .::LclRotation, .::LclScaling - apply via .::EvaluateGlobalTransform to get model-space coords
    - Vertex data in FbxMesh.GetControlPoints() - consists of 4-double vectors (FBXVector)
    - Indices not usually accessed so simply - In theory, can do it w/ ..GetPolygonVertices()
    - Normals in ..PolygonVertexNormal(polygon index, vertex index, destination FBXVector)
      * Access w/ nested loop - first over polygons (get count with ..GetPolygonCount), then over indices (get count with ..GetPolygonSize(polygon index))
      * In theory, can do it w/ ..GetPolygonVertexNormals (destination array)
      * OR by ..GetElementNormal().GetDirectArray().GetAt(polygon index)
      * Normals have multiple accessing methods, found in FbxLayerElement::GetReferenceMode():
        + eDIRECT = normals map directly to vertice array
        + eINDEX\_TO\_DIRECT or eINDEX = normals have an index map, which you find in FbxLayerElementTemplate.GetIndexArray()
    - UVs in ?
      * Access w/ GetPolygonVertexUV, similar to normals
    - Materials in FBXNode.GetMaterial(material index); indices w/ ..GetMaterialCount()
    - Textures (FbxFileTexture) attached to Material properties (FbxSurfaceLambert.Diffuse, ..Bump, etc.); you get associated file with GetSrcObject(object index), get indices w/ GetSrcObjectCount; must cast from FbxObject

**COLLADA**

**Procedure:**

* Use Open Asset Import Library, aka Assimp. You do similar prep to the FBX SDK, but it's less of a massive pain to pull the data. Gen a aiScene with Assimp::Importer.ReadFile(full path, postprocessing flags).
  + Flags include things like aiProcess\_Triangulate and aiProcess\_GenNormals (or aiProcess\_GenSmoothNormals)
  + By default, triangle winding is counterclockwise. Inform renderer if it expects otherwise
  + All data is also per vertex - if you have flat shading normals, you will have duplicated vertices. Unclear if you can immediately pass the given buffers, but I don't see why it wouldn't work.
* Meshes will be in mMeshes; form vertices from mVertices (position data), mColors (color data) and mNormals (normal data).
  + If color isn't provided, set to Colors::White.
  + If normals aren't provided, set to null? A bad default value? I dunno, we need to see how lighting is handled.

**Details:**

* + - Meshes in mMeshes
      * Vertices in mVertices, each consisting of 3-float vectors
      * Normals in mNormals, 3-vectors
      * Indices contained in mIndices, which is contained an aiFace struct, faces contained in mFaces
      * UVs in mTextureCoords
        + Dimensionality of UVs in mNumUVComponents
        + Is an array of arrays - mesh can have up to 8 different UV channels
      * Bones in mBones
        + Bone weights in mWeights; each weight consists of an index to the affected vertex and the weight's strength
      * In general, array members also have their count in a var - mVertices has mNumVertices, for instance.
    - Textures in mTextures
      * Dimensions in mHeight, mWidth
      * Raw data in pcData; if mHeight == 0, data's compressed, figure out format with achFormatHint. If it's compressed, the renderer will need to know, or you'll need to decompress it into the exported file
    - Materials in mMaterials
      * Get property with Get(name of property, type, index, output buffer)
    - Animations in mAnimations
      * Duration in mDuration, measured in ticks. Tick rate in mTicksPerSecond
      * Bone based animations in mChannels
        + Complex; look at the docs @ <http://assimp.sourceforge.net/lib_html/structai_node_anim.html>
      * Vertex based anims in mMeshChannels
        + Mesh name in mName
        + Keyframes in mKeys
    - Count for all prior elements in mNum\* (e.g., mNumMeshes)
    - Existence of element found with Has\*

In any case, the idea is that you import the file, either directly parse (can be very slow if the geom's complex) or read into a POD struct that you can then write straight to disk - with latter, you can then read into a buffer and then cast it back into a struct. Binary option requires you check endianness, however! Maybe make a tool to write in whatever endianness you need.

**Model Format at Runtime**

Models consist of a group of ModelUnits, which are a Geometry unit combined with a Material. Materials contain several color properties (all Color objects):

* Diffuse
* Specular
* Emissive
* Ambient? (Unsure if this will be kept, may just use diffuse)

They may also contain a single texture reference for *each* of these properties. In the future, this can be expanded to texture stacks.

**Model Format on Disk**

Models (.lmdl files) are **little endian**, and start with a model header:

1. Signature 0x4C4B4D44 (U32)
2. Version number - at present, 200, or 0xC8 in hex (U16)
3. Number of model units (U16)
4. Offset to start of mesh headers (U32)

5-7. The dimensions of an axis-aligned bounding box (AABB) that encloses all of the model's vertices (F32).

8. The radius of a bounding sphere that encloses all of the model's vertices (F32).

This is then directly followed by a list of model unit headers:

1. Signature 0x4C4B4D55 (U32)
2. Offset to model unit's data (U32)
   1. = Last header's item 2 + this header's item 3.
   2. First offset = model header's item 4.
3. Total size of model unit's data (U32)
   1. = sum of items 4, 5, 13 through 16
4. Size of vertex data, in bytes (U32)
   1. = item 6 \* sizeof(Vertex)
5. Size of index data, in bytes (U32)
   1. = item 7 \* sizeof(U32)
6. Number of vertices (U32)
7. Number of indices (U32)
8. Number of UV channels (U8)
9. Material: Diffuse Color (Color = Vector4)
10. Material: Specular Color (Color)
11. Material: Emissive Color (Color)

12-15. The length of the GUIDs for the textures corresponding to each material channel, including the null terminator (all U32). If a length is 0, that indicates no texture is used for that channel. #16 is the GUID for a normal map, if the mesh uses one.

16. Offset to next mesh header; a 0 value indicates this is the last mesh in the model (U32)

This is then followed by the model unit data, placed in sequential order. Each block of data consists of the material's data:

1-4. GUIDs to textures. Each is null-terminated; in theory, all you'd need to do to rebuild a GUID is pass the pointer to the raw data plus the offset to the model unit plus the length of each GUID before this one.

The geometry data follows:

1. A block of vertex data
2. A block of index data

If any texture GUIDs fail to resolve, resolve them to some default failsafe texture.

**Future Processing?**

Creating triangle strips from mesh?