Appendix 2: Reading From The PackFile

Reminder

Package
The Package describes the entire PackFile. All the arrays within it only describe a type of data and where it is in memory. In other words, you may iterate through the meshes and find the location of their buffers in the gpu_data memory.
Header -> The header simply tracks the version and buffer counts
Array<Buffer> -> A Buffer describes information to retrieve the buffer from the gpu_data
Array<Mesh> -> A Mesh describes information of a given mesh (vertex buffer, index buffer, etc.)
Array<Material> -> A Material describes its properties as well as the textures it will use
Array<Node> -> A Node describes one or multiple meshes, materials as well as their common transform.
Array<TextureRef> -> A TextureRef only contains the path for a texture to be loaded
Array<Skeleton> -> A Skeleton describes information needed to animate a mesh
void* gpu_data -> This is the start of memory where all the information described by the above arrays is contained. Essentially, loading from a PackFile means interpreting data store from here correctly.
size_t gpu_data_size -> The total size in bytes of gpu_data

Foreword

Reading the PackFile data is more straightforward than it seems for simple purposes. Do not be daunted by the code at first glance, this document will help you break it down into separate parts and explain what each part does.

Note that we will not cover the InitAsCube() function as this is a simple port of the code you already built in the *PS5 Hello World Cube* document.

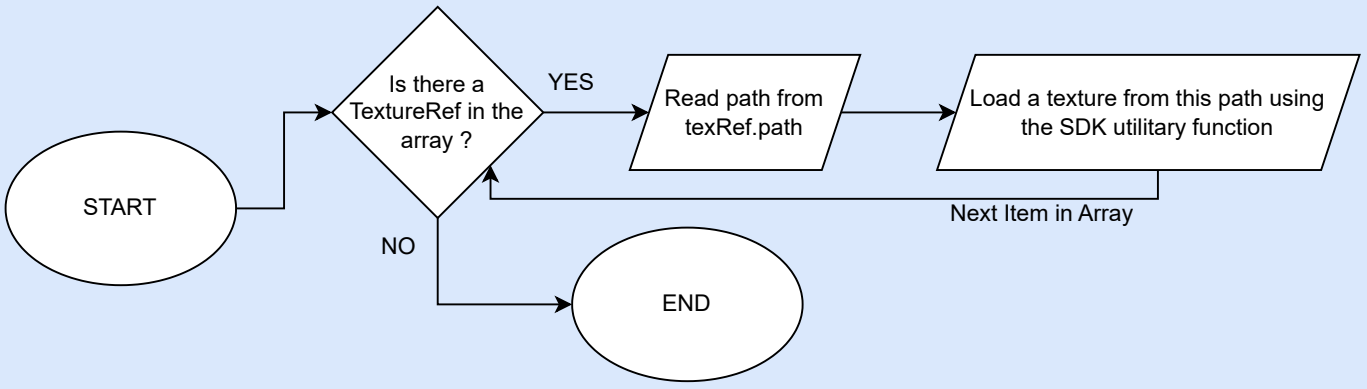
We suggest to have the code alongside this document to better make sense of the subjects we describe.

For our purposes, we do not consider Physical Based Rendering (PBR) materials and model animations. Recall from *Appendix 1* that a model may be described by a combination of multiple meshes. Therefore, we will build a MeshInstance that contains one or many meshes as well. To achieve this task, the InitFromPack() function can be broken down into the following tasks:

1. Load all textures
2. Retrieve the texture index we need for a mesh
3. Retrieve the indices of the mesh
4. Retrieve the vertices of the mesh
5. Create the vertex and index buffers on the PlayStation 5 memory
6. Render the mesh instance (i.e. all the meshes)

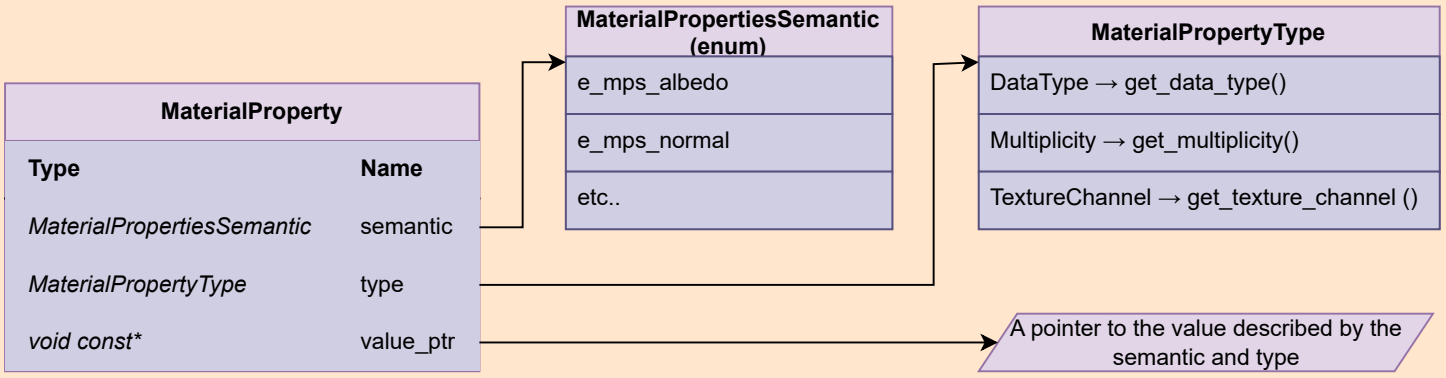
1. Load all textures

This step is rather straightforward. We will simply loop through the array of TextureRefs present in the Package to obtain the paths of our textures. The only trap is to remember that all paths must be preceded with "*/app0/*".

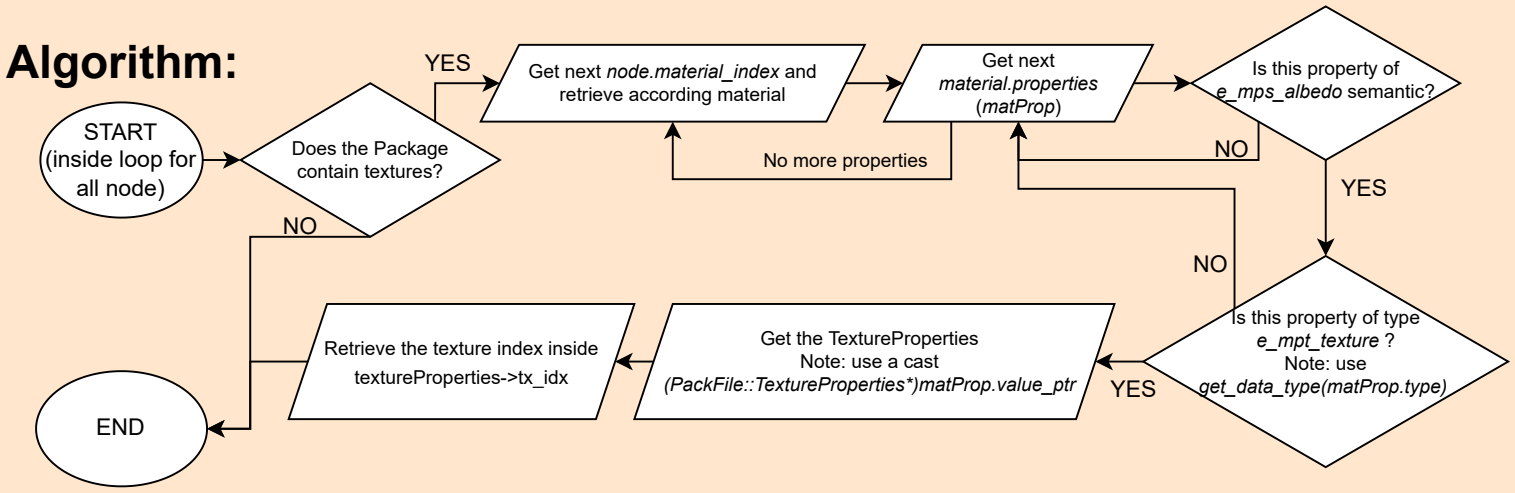


2. Retrieve the texture index from the material

The node stores indices for the materials to use on its mesh(es). Note by definition of the structure, multiple materials may be applied to the same mesh(es). However, for now we are only interested to texture our mesh(es), that is to retrieve the albedo map. So, to achieve this, we will loop through all the materials, based on the indices given on the current node, and determine if a material contains the albedo map based on the *MaterialProperty* information:



After finding the *MaterialProperty* with the correct semantic (*e_mps_albedo*) and the correct type (*MaterialPropertyType::e_mpt_texture*), the texture index is obtainable by casting the *value_ptr* to the according property, in this case *TextureProperties*. This structure defines a member variable *tx_idx* which is the texture index we are looking for!

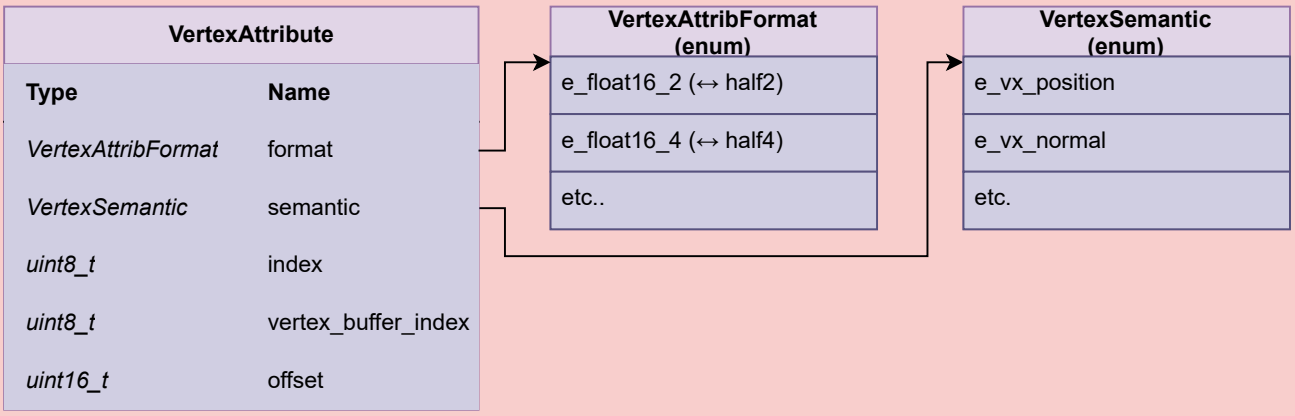


3. Retrieve the indices for a mesh

Like most of data we are dealing with, the indices are stored inside the *gpu_data* memory region defined in the Package. In order to get our indices we need to retrieve a buffer description of the index buffer. This buffer description will tell us how to read from that memory region properly and thus getting our indices. However, there is a catch: the indices may come in different formats: 8 bits, 16 bits or 32 bits. Therefore it is important to account for these format as they define how many bytes an index occupies in the *gpu_data* memory. A *PackFile::Mesh* describes the *index_elem_size* but this is in bits. The buffer description gives us a *stride*, in bytes, that essentially tells us the same information. We therefore used the stride to implicitly convert the indices on copy. The code should be straightforward enough to read: we retrieve the buffer description with the given buffer index and use it to locate the information in the *gpu_data* memory.

4. Retrieve the vertices for a mesh

Retrieving the vertices follows similar logic to the indices, that is we first locate where the vertex buffer starts in the *gpu_data* memory before handling some type conversions. However, there will be a bit more work here.
The mesh contains a set of attributes. These attributes define a section of the vertex type, such as the vertex position or the vertex normal. Each of these two are stored as different attributes. It is our goal to read all the attributes and store our data accordingly.



The algorithm will loop through all the attributes and first check the semantic to ensure correct processing of the attribute's data. After that, the data may come in multiple formats. This is up to our implementation to read from these formats correctly and handle proper conversion. However, in our testing the PackFile converter always produced the same set of formats for each attribute, and therefore we have only accounted for one format. It will be up to you to improve our code to support more formats if need be.

Here are the formats we are handling for each semantic:

Vertex Semantic & Format Relation	
semantic	**format**
e_vx_position	e_float32_3 (32 bit float)
e_vx_uv_channel	e_float16_2 (16 bit float)
e_vx_normal	e_sn_int16_4 (16 bit signed integer)

Note that the position data does not require any conversion as we also consider 32 bit floats. However, the uvs and normals will require some type conversion. We have used a functions provided in the DirectXMath library to handle a conversion to a 32 bit float in both cases. In our code, these function are referred to as *Utils::halfToFloat()* and *Utils::snorm16ToFloat()*.
Reading from the *gpu_data* works in the exact same way as with the indices. However, the position and normals will need an extra transformation to place them in the correct location. Recall from *Appendix 1* that a node defines a transformation matrix for all the meshes it contains. Also recall that the node contains the index to its parent node, which may also be subject to some transformation. Therefore, after reading the position data, we first transform it according to the parent matrix, followed by a transformation of the current node's matrix. This should result in placing the mesh in the correct location in space. Note that a the time of writing this document, we are unaware if the transformation of a child node is based on the parent's or based on the origin. We organised our code considering it is based on its parent. The same logic applies for the normals. However, just like in shaders, we will only need to consider the rotational part of the transformation, that is the first 3x3 section of the matrices. This is why the position uses the function *Utils::MatrixMul()* while the normals use the function *Utils::MatrixMul3x3()* for according operation.

5. Create the vertex and index buffers on the PlayStation 5 memory

Vertex and index buffers are created in the exact same manner as the original cube. Please refer to the *PS5 Hello World Cube* document if you struggle to understand the code. The only difference here is that each mesh that is part of the mesh instance will have its separate vertex and index buffer. We have chosen this design for simplicity, but feel free to improve our code by compacting all the buffers into one and produce a single draw call!

6. Render the mesh instance

Finally, we can render the mesh! Have a look at the render function. Based on our loading method, this become rather easy. We will loop for all the meshes this instance contains and render each one separately. The texture of a mesh is obtained via the texture index we have stored and the vertex and index buffers are also accessible on that mesh. Therefore, we simply set all that data on the pipeline and issue a draw call.

Final Overview - InitFromPack()