Simple Model Loading

This document will cover the different steps to load in a model from scratch on the Sony PlayStation 5. The resulting code from the *PS5 Hello World Cube* demo is taken as a basis for our implementation, so make sure you have done that first. The goal is that by following through the steps presented here you would end up with a sample application that can load simple models. Note that this implementation has been tested for simple models (see results screenshots near the end) and that this implementation has flaws that we purposely leave out for you to modify when required. All details about these further modifications are found in the corresponding section at the very end of the document. A set of helper tools are provided with this document to help with the different conversions we will go through.

Tasks Breakdown:

- Converting a texture to GNF format using the sample texture converter
- Modifying the pixel shader to consider a sampler and a texture, create a simple sampler on the C++ side.
- Downloading a model and exporting it from Blender
- Converting a model to a PackFile using the sample converter
- (Bonus) Using a GP5 file for data transfer
- Adding the PackFile parser library to our code & loading the PackFile
- Writing a custom MeshInstance class that will be parsing the data from the PackFile so that it can be rendered

1. Converting a texture to the GNF format

As not every model can come with a texture, we will consider a default texture that will be applied in that event for simplicity. The original image we have chosen for this purpose can be found here. However, textures are expected to be in the Sony proprietary GNF format. For shorts, it is "*designed to contain runtime ready texture data*" and metadata. You would be able to use traditional DDS or PNG formats if you wished to, but you would be required to convert them to runtime-ready images using the AgcTextureTool in runtime. Converting your images to GNF files offloads this work offline and allows you to directly load the data in your application instead of having to handle the conversion.

In the helper folder browse the tools folder and find the texture converter. The executable *texture_tool_sample.exe* has been compiled from the sample provided with the SDK ($(SDKDir)6.000\host_tools\samples\texture_tool). We will be using the windows command prompt (type in "cmd" in the windows search) to use this tool and convert the downloaded texture. The command format is as follows:

texture_tool_sample.exe <input path> <output path>

You can set the path of the helper folder in front of the exe name in this command or change directory to it. For example, the following commands allowed us to convert our default texture to our project directory:

```
>cd C:\Users\Documents\GitHub\PS5\ETC_StarterCube\model loading helper folder\tools\texture converter
```

>texture_convertex.exe C:\Users\SDI\Downloads\DefaultTexture.png
C:\Users\SDI\Documents\GitHub\PS5\ETC_StarterCube\Cube\Cube\data\ps5_assets\DefaultTexture.gnf

```
Microsoft Windows [Version 10.0.22621.1105]
(c) Microsoft Corporation. All rights reserved.

C:\Users\SDI>cd C:\Users\SDI\Documents\GitHub\PS5\ETC_StarterCube\model loading helper folder\tools\texture converter

C:\Users\SDI\Documents\GitHub\PS5\ETC_StarterCube\model loading helper folder\tools\texture converter>texture_tool_sampl
e.exe C:\Users\SDI\Downloads\DefaultTexture.png C:\Users\SDI\Documents\GitHub\PS5\ETC_StarterCube\Cube\Cube\data\ps5_ass
ets\DefaultTexture.gnf

C:\Users\SDI\Documents\GitHub\PS5\ETC_StarterCube\model loading helper folder\tools\texture converter>
```

Note that we have created a data folder and a ps5_assets folder in our project directory to receive all assets that we will use in this procedure. Executing both above commands should result in having the DefaultTexture.PNG converted to GNF and exported directly to our new folder.

2. Texturing the Cube

Now that we have a default texture exported, let's try to use it to texture our cube. Just like with DirectX, we will need to use a texture resource and a sampler. In the code helper folder, you will find *Utils.cpp & Utils.h* These files define a set of helper functions that we will use throughout this guide. Simply copy the files into your project's directory and add them to the Visual Studio project. We will load the texture and create a sampler using the following code outside of your main loop (we do not want to load the texture every frame!):

```
#include "Utils.h"
…
void agc_main()
{
…
// Load the default texture
sce::Agc::Core::Texture tex = {};
error = Utils::loadTexture(&tex, "/app0/data/ps5_assets/DefaultTexture.gnf");
SCE_AGC_ASSERT(error == SCE_OK);

// Create a default sampler
sce::Agc::Core::Sampler sampler = {};
sampler.init();
        …
}
…
```

Note that if you completed step 5 your input path would differ from the above example: "/app0/DefaultTexture.gnf". Try running the above code and check that your texture is loading properly.

Once that is done, we need to supply both the texture and the sampler to the graphics pipeline. Add the following modifications to stageBinder:

```
…
void stageBinder(…, sce::Agc::Core::Texture* tex, sce::Agc::Core::Sampler* sampler)
```

```
{
…
// Set constant buffers
…

        // Send the texture and sampler to the pixel shader
psBinder.setTextures(0, 1, tex).setSamplers(0, 1, sampler);

// Draw
…
}
…
void agc_main()
{
…
// (In main loop)
stageBinder(…, &tex, &sampler);
…
}
…
```

Finally, we need to modify the pixel shader to sample the texture. Modify your *pix_p.pssl* shader with the following changes:

```
struct V2P
{
…
};

Texture2D<float4> inputTexture;
SamplerState sampler;

[CxxSymbol("Shader::ps")]
float4 main(V2P input) : S_TARGET_OUTPUT0
{
return inputTexture.Sample(sampler, input.uv);
}
```

And that's it! You should obtain the below textured cube!

--- Add Screenshot

3. Downloading and exporting a model from Blender

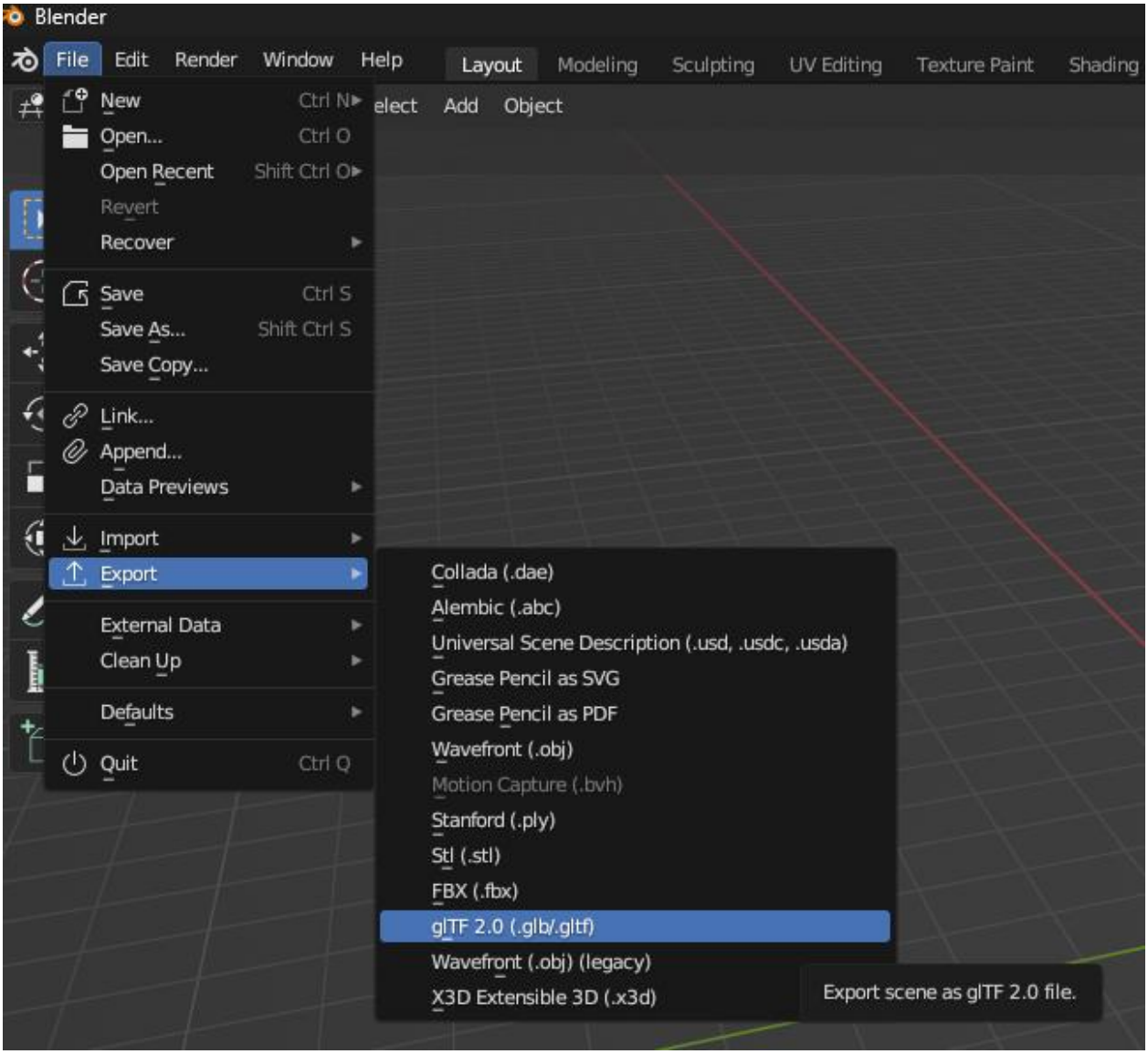
The next step will be to choose and download a model from the internet. The compatible formats we can use are GLTF, FBX and OBJ. Additionally, we can also consider BLEND files as they are scene files for Blender, which can then be exported to a GLTF model from there. We chose this model, as it provides a

Blender scene with textures. Remember that if your model does not have textures we will apply the default texture exported in the preceding step.

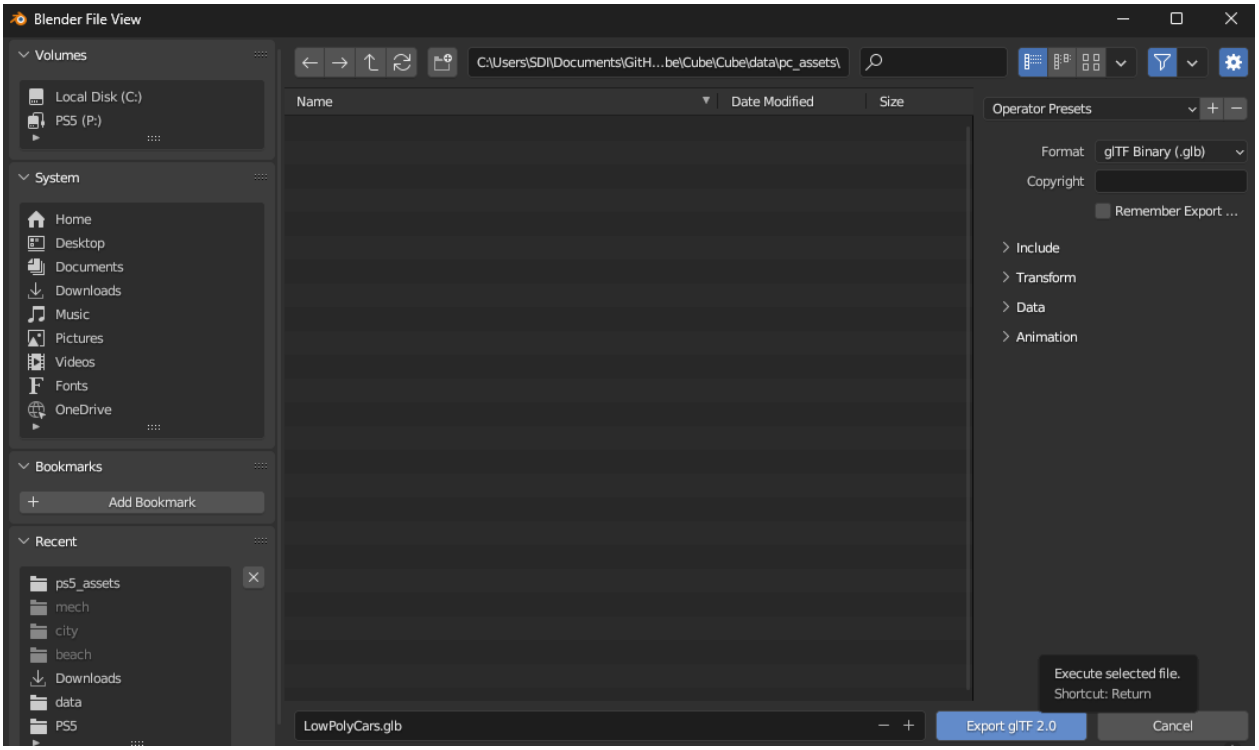
Opening the BLEND file in blender should display the model fully textured when selecting the *Texture* option in the viewport shading. If you do not see textures appearing on the model you should probably consider downloading another model or ask an artist how to bind the textures on your model properly.

From there, select:

File > Export > glTF 2.0 (.glb/.gltf)



On the popup window, make sure to export it to a sensible location (in our case the data\pc_assets folder). On the right-hand side, you will find that you can export to three different formats: glTF Binary, glTF Separate & glTF Embedded. You can choose any format you wish.



Now depending on the format you chose, an additional step is required to supply the textures correctly.

- glTF Binary & glTF Embedded:
  - Locate the PNG textures from the original folder you downloaded from the internet. Copy the textures and paste them into the same directory where you exported the glTF model.
  - Rename the texture files with the name of the glTF exported model + __bufferN, where N is the texture number. For example, our copied textures have been renamed as follows:
    - Car Texture 1.PNG → LowPolyCars__buffer0.PNG
    - Car Texture 2.PNG → LowPolyCars__buffer1.PNG
- glTF Separate
  - After export, you should find a glTF file (.gltf), a binary data file (.bin) and the textures in the previously selected folder.
  - If any texture contains spaces in their name, you will need to open the glTF file (.gltf) in a text editor and replace with a space character all the occurrences of the characters %20. For example, in our case the following entries would be changed:
    - “uri” : “Car%20Texture%201.png” → “uri” : “Car Texture 1.png”
    - “uri” : “Car%20Texture%202.png” → “uri” : “Car Texture 2.png”

You should now be ready for the next step!

4. Converting the model to a PackFile

The Pack format’s primary utility is to play into the strengths of the SSD in the PlayStation 5. In fact, incredibly fast loading and making use of hardware decompression is a pillar of PS5 development. Therefore, the goal is to load the mesh data as efficiently as possible into memory without lengthy post-load processing. Thankfully for us, the SDK provides us with a tool that will handle the conversion directly

($(SDKDir)6.000\host_tools\samples\pack_converter), just like the texture tool. Recall from step 3 that the supported model formats are GLTF, FBX and OBJ. Future versions of the SDK may support other model formats, you can check that be opening the readme_e.html on the pack converter sample directory (note that the above directory is on SDK 6.000).

Using the converter will require us to use the command prompt with the following command:

pack_converter.exe --root <output directory> <input files>

The argument *--root* specifies that the next argument will be the path in which to output the files. Note that you can supply multiple input files as well, it will simply create a pack file for each input provided. Below is an example command we used to convert the LowPolyCars model from glTF (.glb/.gltf) to a pack file:

>cd C:\Users\Documents\GitHub\PS5\ETC_StarterCube\model loading helper folder\tools\packfile converter

>pack_converter.exe --root C:\Users\Documents\GitHub\PS5\ETC_StarterCube\Cube\Cube\data\ps5_assets C:\Users\Documents\GitHub\PS5\ETC_StarterCube\Cube\Cube\pc_assets\LowPolyCars.gltf

After running the second command, no warning or error should be displayed if successful. A typical source of error in our experience was supplying the textures incorrectly, as described at the end of the previous step (Exporting from Blender). We should now have obtained the *LowPolyCars.pack* file in the *ps5_assets* folder!

5. (Bonus) Using a GP5 to transfer data

In Visual Studio, navigate to:

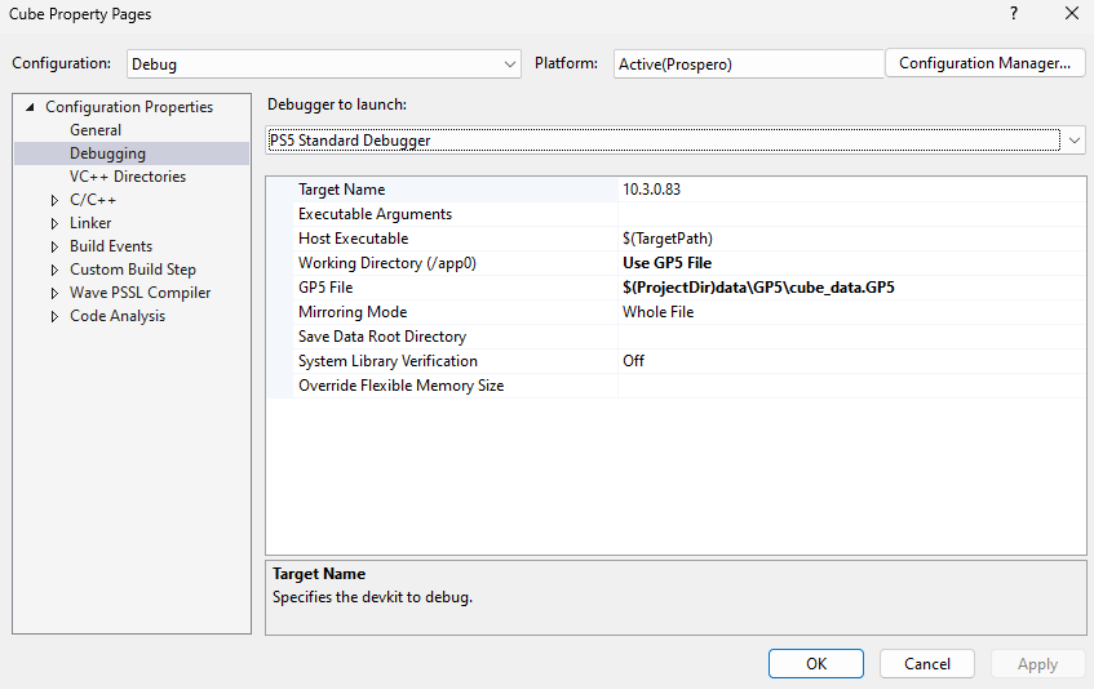
Project > Properties > Configuration Properties > Debugging

Locate the working directory option. This defines the workspace, that is the area that is used during development to contain the files and directories that are available to the running process in "*/app0/*". By default, it should be set to *Use Local Path*, but there are several options to choose from:

Use Local Path	This option means that all the files present in the local path will be accessible for transfer to the PlayStation 5 as they are needed based on the mirroring mode selected. In other words, the PS5 "/app0/" directory is mapped to the PC local path, set in the "Local Path" option in the properties window, and data can be transferred and used directly from it. This mode is very handy for quick development or testing purposes.
Use WorkSpace	This option will use a workspace created on the PlayStation 5 to directly load the assets. This workspace can be created using the WorkSpace Exlporer. The main advantage of using a workspace, in comparison to the other options, is that the files will already have been transferred to the PS5. Therefore, you would not be required to wait for all your assets to be transferred when testing your game, if they haven't already!
Use GP5 File	A GP5 file describes how files are mapped from the host PC file system to "/app0/". The main benefit is that this file can be inputted to the tools when packing the final game for release as a standalone without any changes.

For our purposes, we will demonstrate how to set up a GP5 file so that you can use it in the future when releasing your game. In the same property window, make sure to set the working directory to *Use GP5*

File and in the option underneath type in a path to the GP5 file you will create. In our case, we will later create a file called *cube_data.GP5*, and therefore set the path in the creation directory $(ProjectDir)data\GP5\cube_data.GP5.

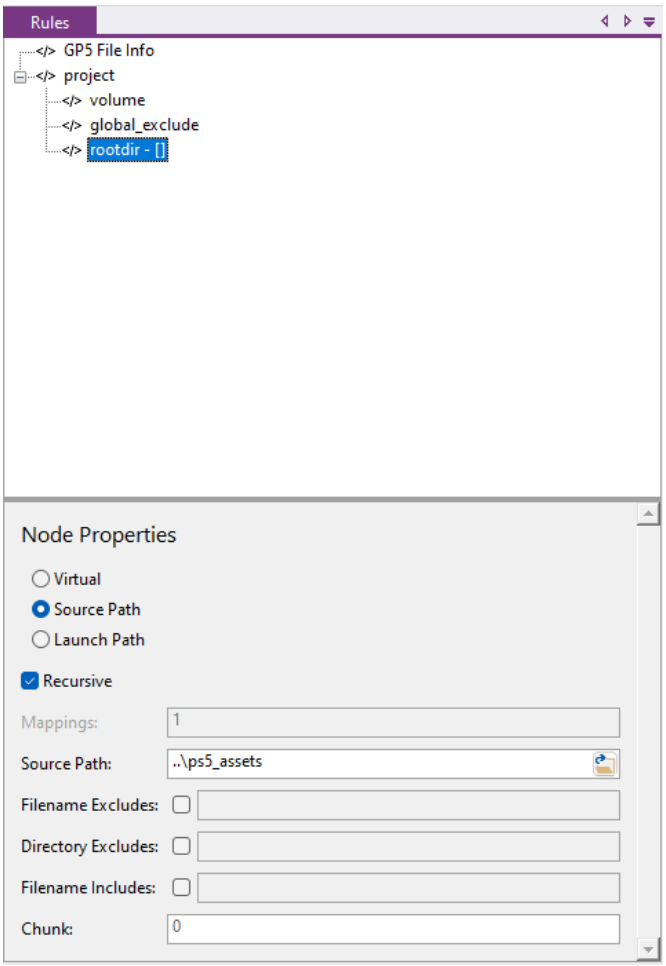


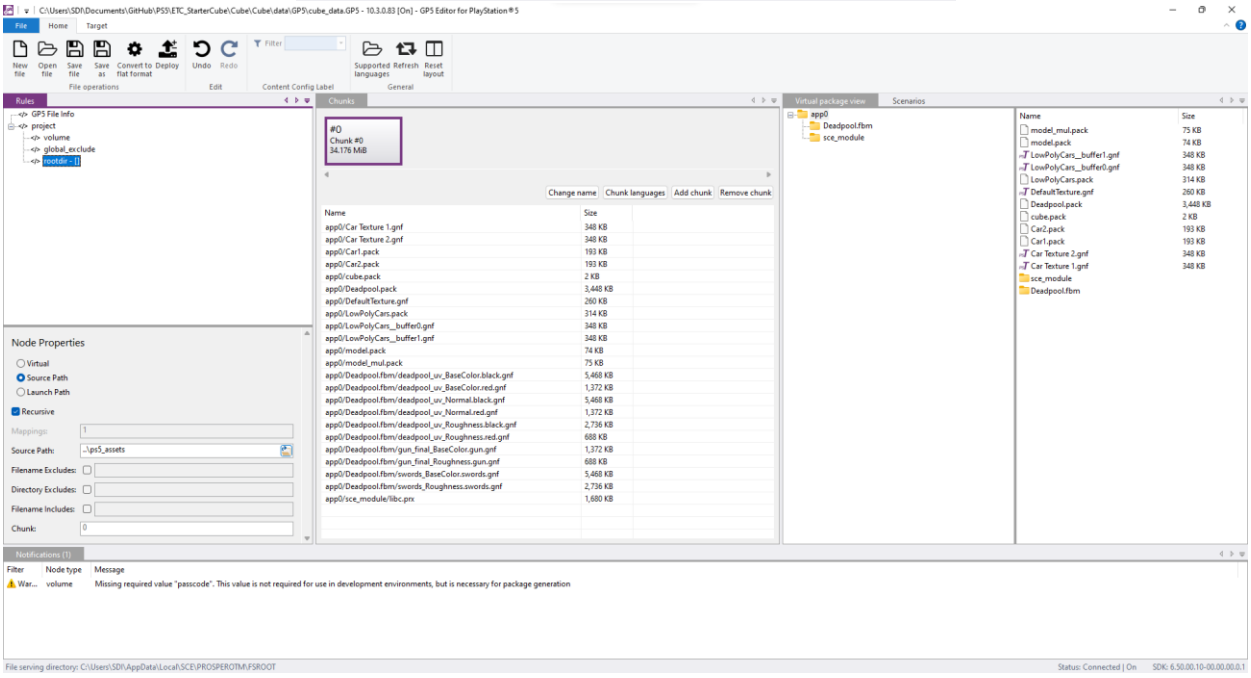
Creating the GP5 file:

Now to create the GP5 we will need to use the GP5 Editor application. It should be accessible from the windows search or from the Apps tab in the Target Manager. Open the application and click on *New File* on the top left corner. You will then be prompted to choose the directory in which to save the GP5 file, make sure to match what we previously set in the project properties. Then, you can immediately click on the *Save File* icon to save your file in this directory. On the first time, you will be prompted to input the name of your file, so once again make sure to match your input in the project properties.

Adding the assets:

The GP5 file is written in an XML format underneath, which means that in the *Rules* window you can find a set of nodes and add properties there. The *GP5 File Info* node allows you to select the directory where your ELF application is exported, though this is not mandatory for our purposes as Visual Studio covers that for us. The main node of interest will be the *rootdir*, as we can set the *Source Path* to our *ps5_assets* folder. That is, by doing so, all the files present in this folder will be automatically considered in our GP5 file. See the below screenshots. After that's done you should be able to see your pack files and GNF textures in the *Chunk #0* window of the GP5 Editor.





Modifying the Post-Build event:

As our application needs to use the SCE library to run, we need to modify the original post-build event to copy the library into our *ps5_assets* folder instead, so that the GP5 file can register it and consequently the PlayStation can access it. In order to do that, navigate to:

Project > Properties > Configuration Properties > Build Events > Post Build Event

Replace the path set in the command line to your directory with your assets (pack files, GNF files). In our case, the final command was:

```
xcopy /Y /I "$(SCE_PROSPERO_SDK_DIR)\target\sce_module\libc.prx"
"$(ProjectDir)data\ps5_assets\sce_module\"
```

And that's it! If all done correctly, your project should now be set up to access data from your chosen directory thanks to the GP5 file. Try running the code, if nothing breaks then your project is successfully using a GP5 file!

6. Loading the PackFile

Now that the model has been converted into a pack file and is accessible on the PlayStation 5, the next step is to load it. The SDK provides a library that will help unpack the data into our application's memory. The helper folder we provided contains a precompiled version of this library, but you can otherwise compile it yourself from $(SDKDir)6.000\sample_code\common\include\pack. Paste the compiled library *libScePackParser.a* in the same *Prospero_Debug* directory where your application's ELF is. Then, paste the *helper code > libScePackParser* folder into your project directory, so that we can include it in our code.

Thanks to this library, loading the pack file is rather straightforward:

```
#include "libScePackParser/pack_file.h"
#pragma comment(lib, "../Prospero_Debug/libScePackParser.a")
…
void agc_main()
{
        …
        // Load the pack file (this is outside the main loop)
        PackFile::LoaderOptions options = {};
        PackFile::Package pack = {};
        error = PackFile::load_package("/app0/LowPolyCars.pack", pack, options);
        SCE_AGC_ASSERT(error == SCE_OK);
        …
}
```

Try running the above code snippet, replacing the pack name with your custom one and see if it loaded correctly!

7. Creating a mesh from the PackFile

The final step will be to interpret the packfile correctly so that we can create a mesh out of it. Remember that a packfile, for our purposes, is essentially a model file converted into a more efficient format for the PlayStation 5.

Refer to **Appendix 1** given alongside this document to understand the structure of a Package object described in a PackFile. This object holds all the data for the model we wish to load. The appendix walks you through the data types and how the overall Package is structured. Refer to **Appendix 2** to see how the Package and the relations between its inner data types are used to load the mesh data into a MeshInstance object. The main challenge is to handle the conversion of data types. For instance, the UVs may be supplied as 16 bits floats, but we require 32 bits float for consistency. It is therefore necessary to check the data type of the UVs and handle proper conversion.

You can use our implementation available in the helper code folder inside *Mesh.h* & *Mesh.cpp*. Feel free to have a look at the appendices in greater detail and our implementation to understand how the model is being loaded and write a loader yourself (warning: it will take some time!). Otherwise, simply add these files to your project and let's take it from here.

Our MeshInstance object provides a function to initialise it from a PackFile. This has been done for our own convenience, as loading a model then becomes straightforward:

```
#include "Mesh.h"
…
void agc_main()
{
        …
        // Load the pack file (this is outside the main loop)
        …

        // Initialise the Model from the PackFile
        MeshInstance model = {};
        model.InitFromPack(pack);

        ...
}
```

Note that our model loader will throw exceptions if the supplied PackFile has unsupported data types. Thankfully however, the vast majority of models exported following our procedure will work without causing issues. If your model threw an exception in our loader, we encourage you to check the difference in data types in the assertion and improve for proper conversion.

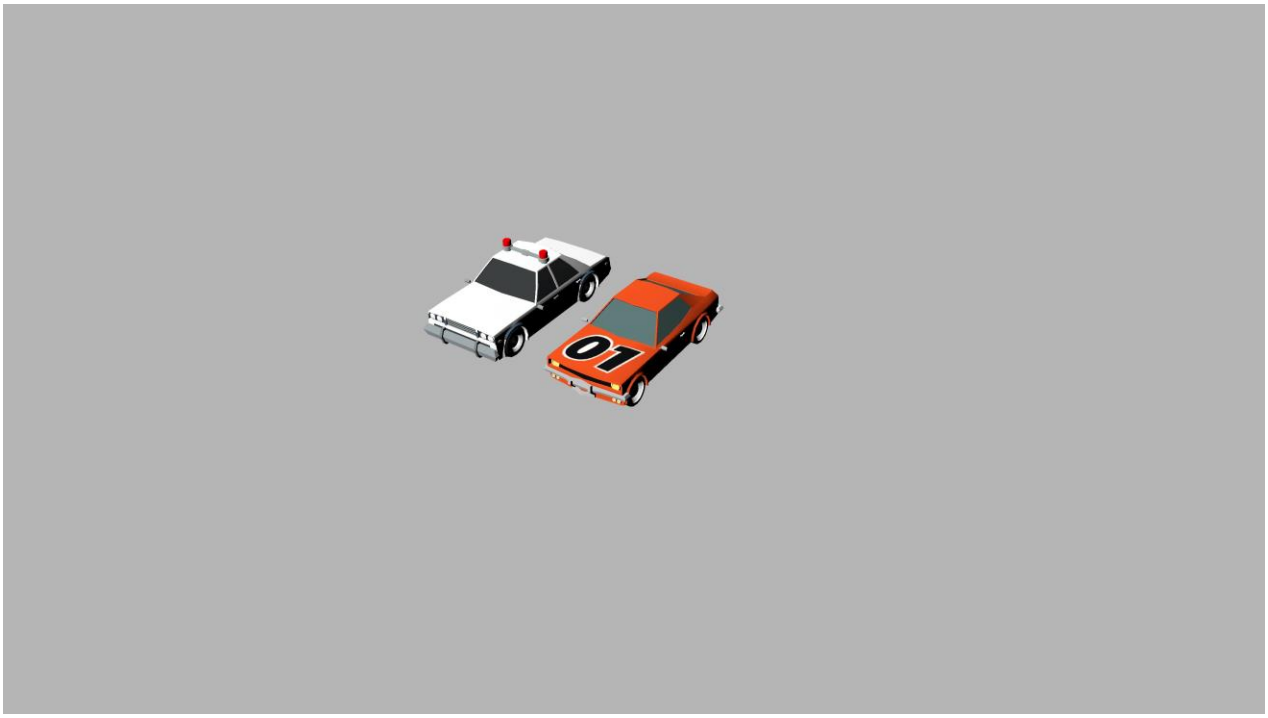
Our MeshInstance object also contains its own render function that essentially replaces the stageBinder function used until now. The main difference is that we supply a single draw call with dcb->*drawIndex()* for each mesh. This is much more efficient than the previous setup that was dispatching a draw call for each triangle and will require a lot less memory.

Therefore, to render your model simply replace the *stageBinder()* call with the model render function in your game loop:

```
…
void agc_main()
{
```

```
        ...
        // Render the mesh (this is inside the main loop)
        model.Render(&dcb, &sb, gs, ps, &PassCB, &ObjCB, 0, &sampler);
        ...
}
```

Et voilà! You can now try to run the code and see your model being rendered on the screen by the PlayStation 5! If you have made it until here, well done! Here is our result with a simple directional lighting applied on the pixel shader:



8. Sources of error and continuing things further

If you have played around with the model loading code, you would notice that our implementation presents flaws. Here are a couple we identified, have a go at fixing them if you wish to:

- Not enough handling of type conversion for vertex data
- Only considering up to 1 parent in the transformation matrices
- Not considering material properties other than the albedo map