Skateboard Engine

Progress Report 29/08/2023

Justin Syfrig

The purpose of this document is to give an overview of the state of the *Skateboard* engine as I am leaving the project by the end of my contract with Abertay University, discuss potential and known issues, as well as where to take things further. This document was **not** made to introduce you to the engine, but rather assumes that you, the reader, have already used the engine and had a look at the code base. Please keep in mind that the work undertaken until here has been made by recent graduates (myself and Rhys Duff), and therefore we can in no circumstances guarantee industry standards and quality of code because of our inexperience. However, we did some research towards these standards and successfully developed a working cross platform codebase (Windows & Playstation 5).

Current State of the Engine

As of today, the engine is ready to use for graphics applications. It is however not yet ready for game development. *Table 1* summarises what rendering functionalities have been implemented in the engine so far. Please note that I will not cover the entity component system known issues and implementation details as I have barely touched this side of the engine. More details on the Scene class & ENTT will follow in a separate report produced by Rhys Duff before he leaves.

	Windows	Playstation 5
Pipelines		
Supported Graphics API	DirectX 12	AGC
Rasterization (Including hardware instancing)	✔	✔
Mesh Shading	✔	✗
Ray Tracing	✔	✗
Rendering Buffers & Objects		
Frame Buffers (Render Textures)	✔	✔
Default Buffers (GPU Only)	✔	✔
Upload Buffers (CPU & GPU)	✔	✔
Unordered Access Buffers	✔	✔

Textures	(.dds only)	(.dds, .png, .jpg, .tga, .bmp)
Byte (Raw) Buffers	✔	✗
Vertex Buffers	✔	✔
Index Buffers	✔	✔
Bottom Level Acceleration Structure	✔	✗
Top Level Acceleration Structure	✔	✗

Table 1 - Summary of the engine rendering functionalities

From the above table you can observe that the rasterization pipeline is considered to be fully functional on both platforms. While this is not necessarily true at the time of writing this document as minor adjustments are still required, these issues will be tackled by Rhys Duff (see section *Current Work in Progress*).
Essentially, the engine offers the ability to use all the legacy graphics techniques, such as rendering to textures for post-processing effects or using the tessellation pipeline. Note that tessellation will work differently on Playstation 5 and you should be aware of that when creating your PSSL shaders. Additionally, the engine offers newer technologies (for now on Windows only) such as ray tracing or mesh shading. Mathematics in this engine are done using the GLM libraries. These offer cross-platform mathematics unlike DirectXMaths which is part of the Windows SDK. Using any mathematical function will require the use of the *glm* namespace as seen on various source files (example: *glm::lookAtLH*). However, for consistency with HLSL and PSSL shaders, we have created several typedefs (inside *src* > *Skateboard* > *Mathematics.h*) that may help working with these data types.

Current Work in Progress

Before leaving the project I pushed the maximum of Playstation 5 functionalities possible in the time frame. My goal was to have a complete support of the rasterization pipeline, and this target has almost been achieved in the sense that only minor adjustments are required. The rasterization pipeline class is considered to be ready, as well as most of the different API buffers implementations. R.Duff will be fixing the following known issues by the end of his contract in order to guarantee cross-platform compatibility:

- Adding a gpuSyncEvent on AGCFrameBuffer::Unbind() as I had forgotten about the parallelisation of Playstation rendering in contrast to the linear execution in DirectX over a draw command buffer or command list. The AGCFrameBuffer class is otherwise complete (except for the next bullet point) and should result in the same operations as on Windows.
- Ensuring that the AGCFrameBuffer class creates a render target with the correct format given by the FrameBufferDesc on creation. Same ideas for AGCUAVs and AGCTextures.
- Adding an instancing draw call for Playstation 5. All the instancing data should already exist as instancing is present on Windows. It should be a simple matter of

emitting the correct draw call. From my previous, very quick research on the topic, I believe that the instance index can be offset on the draw call or by using another function on the dcb just like Vulkan, which would avoid the root constant we have been using in DirectX. On this same note, I would like to abstract this root constant again (I've changed it to be inputted by the user on the pipeline description, but I think this is a bad idea as other APIs do not need it (and one day DirectX may not need it too)).

- Adding a cross-platform input system, such that a camera can be controlled on both platforms with the same code.
- Adding timing code on the PlaystationPlatform class to produce and retrieve deltatime on demand. This could be simply made by using Frank Luna's Timer class as we did for Windows, and simply abstracting the timing functions to the platform.
- Testing the vertex buffer is properly provided to the PSSL vertex shaders by the AGCPipeline::SetVertexBuffer(). At this time, the vertex buffer location is identified to be the last resource sent in the shader as this has seemed to be true for all SRT Signatures I have tested. However, I would be grateful if this could be crash tested further with lots of empty buffers and such in the SRT and making sure that the vertex buffer is still being correctly sent.
- Continuing his ongoing work with memory management, allocation and release for maximum efficiency on the Playstation.
- Releasing buffers manually if desired. A Release() interface has been created for manual release of GPU resources, but as of now remains empty. This will be either continued or removed.

Other Known Issues

This is a little bit of an info dump of everything I could think of, out of order, so apologies if this is a tad messy to follow. Details for Mesh Shading will come from Rhys Duff in a separate report.

- Playstation is missing Mesh Shading, as well as Ray Tracing support. These should be implemented to continue the cross-platform compatibility.
- The Playstation Debugger may hang Visual Studio after a very minor modification in the code base. When that happens, there is no other option than force quitting Visual Studio and relaunching (which will work 100%). The reasons for this hang are unknown and I am unsure if it has to do with the engine code, as I have also experienced it with samples on SDK 7.
- Windows can only load DDS texture files. It would be a good idea to improve this by considering PNG, JPG, TGA and BMP for a good enough variety of formats. Students are more likely to use the PNG and JPG formats, so these should be accounted for at a minimum. The Playstation 5 texture converter will automatically handle all the formats listed above as a post-build event of the GameApp project, so there is no need to worry about this platform.
- Bottom level acceleration structures for ray tracing only consider one single mesh. When dealing with models, this will become an issue as a model may contain many more sub-meshes. A change may be required to upgrade the bottom level acceleration structure to accept one or many meshes as well. This would also allow custom implementation to group the many (or all) static meshes of a scene into a

single (or couple of) bottom level acceleration structure as it is recommended by the Nvidia and DirectX teams.

- The acceleration structures are always built to prefer fast tracing. This could become an issue when dealing with model animations as you would prefer a fast build. For now this is correct and efficient as a user is not likely to update many meshes at once and geometry remains mostly static, but this is a detail to keep in mind when dealing with animations.
- Premake is still present in the project, but we have deprecated its usage since the Playstation development started because it would remove all the project setup made for this platform. Premake was also found to not be able to recognise HLSL shader libraries and you would have had to manually reset them to their correct shader type after generating the project files. So we elected to stop using it entirely, as the project remains rather compact anyways. Further efforts could be made to modify premake and add these functionalities ourselves, but this is definitely not in the scope of this project.
- All efforts have been made to correctly account for frame resourcing. However, I am pretty sure that some spots may have been overlooked and are giving a change only once every third frame. For instance, in the D3DRaytracingPipeline I have created multiple descriptor tables for the different root signatures and constructed them on initialisation time. While I tested it to be accurate and smooth, I am still not convinced that everything is accounted the way it should. I did not have time to verify this, so if noticeable stutters occur, especially when running at 60 FPS or lower, it could be a good idea to test the different pipelines and capture six frames in a row while constantly updating a mesh, moving the camera, etc. That way it would be relatively easy to identify which buffers are not correctly updating (if any).
- Engine logging does not occur at all on Playstation, the reason being that spdlog, the library used for logging on windows, is incompatible with the platform. I quickly tried to use *printf*, *cout*, or other equivalents as seen on the samples but it did not output anything to the Visual Studio console as expected. I must have missed something little about this, but it could be handy to enable our logging on Playstation.
- The Skateboard::Application::Run() executes an infinite loop that is not being paused or running at reduced rate when the application is not in focus. Both platforms may have the game out of focus and therefore should be accounted for.
- The Windows platform only renders graphics using DirectX 12, but it could be relevant and very useful for students to see how Vulkan can be integrated in this engine as well. Both APIs work with similar ideas, but noticeable differences remain. It could also bring an interest for:
    - Shaders need to be created once per supported API. That is, since only DirectX 12 and AGC are supported at this time, you will need to create a HLSL and PSSL version of the same shader. It could become handy, especially after integrating more APIs, to have a unique shading language that is being converted to their corresponding counterparts.
- The ImGui libraries had to be slightly modified to account for the Playstation support. Right now, these are being cloned from a public repository on my GitHub account. It could be more interesting to switch it to the Abertay SDI as a repository only accessible to the students. That way, you could also add the NDA backends into the repository which would clean the project a bit more, as right now these backends are

part of the *Skateboard* engine repository. Please notify me when you are not using the repo on my account so I can remove it.

Taking Things Further

As discussed until now, the engine is quite ready to be used for graphics purposes and experimentations. However, in order to become a proper game engine, further development on the following key points would be necessary:

- Integrating a physics engine. We recommend Nvidia's PhysX as it is widely used in many game engines as well. We believe that PhysX should be compatible with all platforms, but you could decide to use Sony's physics engine (in samples) as well. Bullet3D is another option, but we do not recommend it as it runs quite badly in debug mode (during the CMP418 animation module in 2022, for a simple ragdoll you would average 30 FPS).
- Offering a default initialisation of graphics. Right now all the graphics must be initialised by the client (as required for the graphics module), but for a games module you would probably want to have some default shaders, materials, shadows, etc figured out and ready to use.
- Model loading. At this time, only simple primitives can be created by the SceneBuilder, with the exception of the dragon model integrated from a custom binary for DirectX mesh shading. Loading models effectively is a notoriously difficult task and so we recommend the usage of an external loader for the engine. We may recommend investigating the DirectX Mesh libraries, as they should provide a way to convert models from various formats and make them compatible with the mesh shading pipeline. Note that using models on the Playstation should work as well using the same libraries of your choice, as it is principally just reading a file and creating vertex data from it. However, if you wish to push things further to play in the strength of the playstation SDDs, you may wish to convert these models using the Sony PackFile converter instead. This software, just like the DirectX Mesh libraries, will load and convert models from various formats to *.pack* files, specifically designed for the Playstation hardware. To my knowledge, this converter will however not create mesh shading ready data, and you will have to implement or use another library like DirectX Mesh to use the Pack data and prepare it for mesh shading.
- Creating a proper level editor with some serialisation, such that students can create their levels in the editor, choose their assets and place them at relevant places and save their work. This could be much more intuitive and better practice than hard-coding assets and transforms.
- Load and render model animations. This would create an enormous experience improvement, but may require a lot of work for its implementation, especially when trying to make the animation system compatible with raytracing (updating/refitting bottom level acceleration structures).
- Using scripting to create game logic. This would transform the *Skateboard* project from a framework to an engine, as students would not have to touch the engine code at all to create their games and rather keep working with different script files that they would serialise as components on their different entities with ENTT. In that sense, the engine would become a complete application that does not necessarily require access to its source for early years modules.

With these systems in place, you should obtain a reasonable engine that would allow you to create simple games. At this point in time, I think that there is still a lot of work! I had a lot of fun working on this project, and I really wish I could have spent a lot more time on developing Playstation 5 graphics, especially PSRaytracing, as it is my favourite activity. I hope you have found this document useful, and if you need any help or have any question regarding past or future development please feel free to email my work inbox justinengineprogrammer@gmail.com and I’ll try my best to get you sorted!