|  |
| --- |
| Skateboard Engine  Progress Report 25/09/2023  *Rhys Duff* |

Contents

[Current State of the Engine 3](#_Toc146622670)

[Remarks on Progress 6](#_Toc146622671)

[Instancing (AGC) 6](#_Toc146622672)

[Memory Allocator & Pool Allocation (AGC) 7](#_Toc146622673)

[Renderer 7](#_Toc146622674)

[Model Loading 7](#_Toc146622675)

[ENTT & The Scene 7](#_Toc146622676)

[Event Dispatcher 8](#_Toc146622677)

[Diagram of the Event Dispatcher 9](#_Toc146622678)

[Animations 9](#_Toc146622679)

[PlayStation® Animations 10](#_Toc146622680)

[Animation Conversion 10](#_Toc146622681)

[Skateboard custom asset type (Animation, Models …) 10](#_Toc146622682)

[Known Issues 11](#_Toc146622683)

[Future Work 11](#_Toc146622684)

[Material System 11](#_Toc146622685)

[Overview 11](#_Toc146622686)

Current State of the Engine

|  |  |  |
| --- | --- | --- |
| **Graphics Engine** | | |
| **Pipelines** | | |
| Supported Graphics API | DirectX 12 | AGC |
| Rasterization | ✔ | ✔ |
| Mesh Shaders | ✔ | 𐄂 |
| Raytracing | ✔ | 𐄂 |
| **Rendering Buffers & Objects** | | |
| Frame Buffers | ✔ | ✔ |
| Default Buffers | ✔ | ✔ |
| Upload Buffers | ✔ | ✔ |
| Unordered Access Buffers | ✔ | ✔ |
| Textures | ✔ | ✔ |
| Byte (Raw) Buffers | ✔ | 𐄂 |
| Vertex Buffers | ✔ | ✔ |
| Index Buffers | ✔ | ✔ |
| Bottom Level Acceleration Structure | ✔ | 𐄂 |
| Top Level Acceleration Structure | ✔ | 𐄂 |
| **Input** | | |
| **Supported Devices** | | |
| Keyboard | ✔ | 𐄂 |
| Mouse | ✔ | 𐄂 |
| Gamepad | 𐄂 | ✔ |
| **Input Binding** | | |
| Action Mapping | ✔ | ✔ |
| Raw Mapping | ✔ | ✔ |
| **Physics** | | |
| **Nvidia PhysX** | | |
| Rigid Body Physics | 𐄂 | 𐄂 |
| Soft Body Physics | 𐄂 | 𐄂 |
| **Events** | | |
| **Application Events** | | |
| App Render | ✔ | ✔ |
| App Update | ✔ | ✔ |
| App Tick | ✔ | ✔ |
| **Platform Events** | | |
| Window Close | ✔ | ✔ |
| Window Resize | ✔ | 𐄂 |
| Window Focus | ✔ | ✔ |
| Window Lost Focus | ✔ | ✔ |
| Window Moved | ✔ | ✔ |
| **Input Events (Key / Button)** | | |
| Key Pressed | ✔ | 𐄂 |
| Key Released | ✔ | 𐄂 |
| Key Typed | ✔ | 𐄂 |
| Button Pressed | 𐄂 | ✔ |
| Button Released | 𐄂 | ✔ |
| Button Down | 𐄂 | ✔ |
| **Mouse Events** | | |
| Mouse Button Pressed | 𐄂 | ✔ |
| Mouse Button Released | 𐄂 | ✔ |
| Mouse Moved | 𐄂 | ✔ |
| Mouse Mouse Scrolled | 𐄂 | ✔ |
| **Gamepad** | | |
| JoyStick Moved | 𐄂 | ✔ |
| Orientation Changed | 𐄂 | ✔ |
| Trigger Pressed | 𐄂 | ✔ |
| Trigger Released | 𐄂 | ✔ |
| **Audio** | | |
| **Features** | | |
| I/O | 𐄂 | 𐄂 |
| Sound Effects | 𐄂 | 𐄂 |
| Audio Playback | 𐄂 | 𐄂 |
| **ENTT** | | |
| **Components** | | |
| Tag Component | ✔ | ✔ |
| Transform Component | ✔ | ✔ |
| Camera Component | ✔ | ✔ |
| Material Component | ✔ | ✔ |
| Static Mesh Component | ✔ | ✔ |
| Static Mesh Instance Component | ✔ | ✔ |
| Renderer Component | ✔ | ✔ |
| Animator Component | 𐄂 | 𐄂 |

*Table 1 - Updated summary of the engine’s capabilities.*

Remarks on Progress

The following document aims to summarise the progress achieved on the Skateboard Engine. Since the last report, development has involved exploring other aspects of game engine development. There is a fair amount to cover, hence the document is split into sections which explore each aspect in fine detail. Furthermore, each section is composed of sub-sections, including the current state, known issues as well as a short conclusive remark featuring suggestions for possible future work. If you have any questions regarding this document and or the Skateboard engine, please feel free to contact me at [purplemoonone@gmail.com](mailto:purplemoonone@gmail.com).

A small update on some of *Current Work in Progress* discussed in the last report; highlighted points regarding incomplete resource descriptions have been acknowledged and solutions have been implemented.

* This was in regard to the *Frame Buffer, UA Buffer and Textures,* which has since been updated and now supports resource descriptions on PlayStation®.

Synchronising work between the GPU and CPU for resources such as the *Frame Buffer* have also been implemented. 

* Work dispatched on AGC shares a similar asynchronous workflow as DirectX 12 and therefore requires syncing api to ensure resources are safely handled between workloads.

## Instancing (AGC)

PlayStation® now supports instancing; it has been implemented as suggested with the existing scene building system. Furthermore, the PlayStation® supports multiple instanced geometry utilising the simple material system to inherit unique properties per instance, such as different textures.

* The instancing system has been implemented using the indirect draw calls and requires indirect argument buffers to be populated. This has already been implemented, allowing the user to make use of the instancing just as available on windows. AGC offers instancing API however I was unable to get this working as expected, so it might be an area to improve in future.

A time manager class has also been implemented on PlayStation®, calculating the elapsed time of the application as well as the delta time between frames. Both have been tested and work as expected and can be accessed on any layer either through the OnUpdate / OnHandleInput methods, or by retrieving the manager from the static platform getter GetTimeManager(). 

* TimeManager includes functionality for calculating true elapsed time with considerations to application paused delta times. However, this requires testing, and currently the application loop does not track paused timings. This could be achieved with the event dispatching system which is discussed below.

A cross platform input system has been implemented for both windows and PlayStation® to utilise. The system is intuitive to use and works by allowing the user to map inputs to functions and member functions. Additionally, the user can change the type of input mapping based on an enum, KeyDown, KeyReleased etc.

* To bind member functions, the user will be required to make use of the #defineBIND\_INPUT\_ACTION(x) which expands to std::bind(&x, this). Parameter ‘x’ is the member function to be bound as is passed in like: *ClassName::MemberFunction*. Note you do not use parenthesis at the end of the *::MemberFunction* as you are binding the address to the function.

## Memory Allocator & Pool Allocation (AGC)

A memory allocation system was implemented towards the start of the PlayStation® development. The allocator is a simple and allocates blocks of bytes in Direct Memory. The allocator requires a total size in bytes in addition to an alignment value. The allocator returns a handle to the block of memory which contains a pointer to the block of memory as well as a struct used for making enquiries into the block of memory.

* Additionally, a memory pool allocator was also implemented, and has been tested with textures which proved successful. Note, for pool allocations, you allocate a fixed size up front on PlayStation® where each sub allocation on the pool requires the same alignment and cannot be removed until the application terminates.

Renderer

Since the last progress report, the rendering engine has remained the same with a few minor improvements to AGC. I'd like to reaffirm what was stated in the previous report that the engine is in a good place for graphics programming with shaders content. It should be noted that the rendering system requires stress testing to ensure it is robust enough for a classroom environment.

## Model Loading

Skateboard now supports model loading on PlayStation® 5 and can parse .pack files in the AGCAssetManager class. The content for parsing .pack files was built upon Justin Syfrig’s model loader walkthrough. The loader can extract vertices, indices, materials and textures in addition to the newly added blend weights and blend indices, which is discussed further in the animation section.

Model loading is supported on windows but is limited to binary files. Additionally, it is possible to generate meshlet models on windows, more details are provided in a separate document.

* One point to note is while texture loading is also supported, currently the only format supported on PlayStation® is R32G32B32A32. The converter expects a format as an argument which matches the input texture. On loading an R32G32B32 texture, the converter simply refused to parse the texture. We can extend the script to account for multiple texture formats.

## ENTT & The Scene

On PlayStation®, ENTT is supported, provided the language used is c++ 17. Just like on windows, the user can create a component for entities by creating a struct in the component header file.

## Event Dispatcher

A simple event dispatch system was implemented into the project. An event dispatcher is a system which allows the programmer to communicate messages to other systems within the engine, allowing the systems to respond to events appropriately. The system is comprised of two parts, an event, and the dispatcher. An event can be thought of as the data which we would like to send to another system whilst the dispatcher is the function which invokes the event. Events are separated into event categories which derive from the event class for specific systems within the engine. A few have already been implemented and these include the following categories.

* NoCategory
* EventCategoryApplication
* EventCategoryInput
* EventCategoryKeyboard
* EventCategoryMouse
* EventCategoryMouseButton
* EventCategoryGamePad
* EventCategoryGamePadButton
* EventCategoryGamePadTrigger
* EventCategoryGamePadJoyStick
* EventCategoryGamePadOrientation

In addition to a category, an event also contains a type. Event types reflect the specific nature of the event and provides more information on what systems the event aims to reach and affect. Currently, the Skateboard engine supports the following types.

None = 0,

WindowClose, WindowResize, WindowFocus, WindowLostFocus, WindowMoved,

AppTick, AppUpdate, AppRender,

KeyPressed, KeyReleased, KeyTyped,

MouseButtonPressed, MouseButtonReleased, MouseMoved, MouseScrolled,

GamePadButtonPressed, GamePadButtonReleased, GamePadButtonDown,

GamePadJoyStickMoved, GamePadOrientationChanged, GamePadTriggerPressed, GamePadTriggerReleased

An example of a typical event capture and dispatch may look like this. For context, let’s look at a WindowResize event. The category for this event will probably be EventCategoryApplication as the window contains visual elements produced by potentially several different layers within the engine, the major system being the renderer. Each layer will need to handle the event according to their purpose and functionality. The renderer will need to resize the back buffers, as the total pixels on the screen has changed. But remember, the rendering system has no idea that the resize has occurred and will continue to render to the previous dimensions.

We can capture the resize event from the WindowProcedure which checks for the user resizing the window in the background. Once we have captured the resize event, it might be tempting to invoke resizing functions from the WindowProcedure which might suffice for a small application. However, the Skateboard engine will continue to grow and further systems will be implemented which may need to account for a window resize. Therefore, we will store the new dimensions in a WindowResizeEvent local object on the stack. Using a call-back function, discussed in detail below, which points to the OnEvent method in our core application, we will pass our WindowResizeEvent into the call-back. Each platform should contain a member variable which will contain a std::function<void(Event&)>.

This call-back function, in essence a function pointer, will allow us to pass data captured from the WindowProcedure back to our core application. The question remains, when does the renderer become aware of the change? Let’s continue, we invoked our call-back, which pointed to the OnEvent method residing in our core application. This OnEvent method is where we can forward the event to higher layers within our engine. This method loops through each layer and invokes an overridden OnEvent function allowing the layer to handle each event accordingly.

If you are not already aware, the renderer in Skateboard is abstracted and can be accessed in our core application through static methods. One method is the Renderer::OnResize method. OnResize will now invoke the platform’s specific resize capabilities. Our DirectX 12 engine will resize the swap chain. On PlayStation®, currently nothing happens, as we are expecting to render to a fixed size screen. There is a Renderer::SetViewport method which will resize a render target, and is expected to be one of the core render targets used in rendering a scene to.

## Flowchart of a Window Resize Event

A screenshot of a computer

Description automatically generated

*Figure 1 – A flowchart illustrating a typical window resize event.*

## UML Diagram of the Event Dispatcher

A diagram of a company

Description automatically generated

*Figure 2 – A UML diagram of the event dispatcher includes some of the derivative classes but not all of them.*

## Animations

At the time of writing this report, animations are not supported entirely. There has been progress in implementing a cross platform animation system which can calculate the bind pose of a skeleton in addition to extract a pose from an animation with respect to the elapsed time of an animation. This requires testing and may not work as expected.

## PlayStation® Animations

The aim of the animation system is to keep it available to all platforms, regardless of the API used on each platform. PlayStation® 5 offers an animation system included within the animation runtime library and is designed to be lightweight, flexible, and efficient. The runtime library offers animation blending and includes linear blending, N-Way blending and additive blending. The library uses a pose stack to push, blend, sample, and pop skeleton poses. Additionally, the runtime library contains helper functions for converting poses from local space to global space as well as a command interface API allowing for offline blend tree conversion into a command list.

## Animation Conversion

Residing in the host tools of the SDK, *edgeanimcompiler* is a tool which converts the COLLADA format into a format which the runtime library can process. Note, the *edgeanimcompiler* can be compiled into a library enabling support for custom tools creation. Otherwise, the tool can be used with the command line and can generate skeletons, animations and or additive animation runtime files.

A screenshot of a computer code

Description automatically generated

Figure 2 - A snippet of pseudo code for the CMD

Loading animations and skeletons is straightforward and follows a similar workflow to other asset loading techniques. Its possible to allocate animation and skeleton data into main memory. The animation runtime uses headers which contain the necessary data to extrapolate skeleton and animation data. The headers are 16 bytes aligned and therefore require an alignment when allocating in main memory of 16 bytes. Extracting skeleton hierarchy and animation keyframes is achieved through the PlayStation® 5 animation tools library.

## Skateboard custom asset type (Animation, Models …)

Currently, development was headed towards converting a COLLADA into .skel and .anim files to be loaded into Skateboard to be converted further into formats readable for Skateboard’s animation system using the animation tool library. While this would be handled on scene loading time, it is a few hoops to jump through…

* Convert COLLADA into .skel and .anim
* Load .skel and .anim
* Convert runtime skeleton and runtime animation using tools lib.
* Extract skeleton and keyframes and store within Skateboard.

An alternative approach could be to ignore PlayStation®’s formats altogether and either use a pre-existing widely available and used format or create our own. Gef used .scn to convert formats into it’s own and is easy to read and parse once converted.

# Known Issues

There are a couple of issues that remain in the framework which were previously highlighted in Justin’s report and therefore I shall not repeat these points. I shall make a list of points that have been mentioned and remain an issue. These are as follows:

* Skateboard AGC is missing raytracing and mesh shaders.
* The playstation debugger will sometimes hang after launching debugging.
* Engine does not use SpdLogger on playstation platform.
* The core application loop does not run at a reduced rate when the application is out of focus

The AGC engine does not implement support for mesh shaders or Ray Tracing yet. While this is not a significant issue for game development, it would be nice to implement to maintain cross compatibility between platforms.

As noted in the previous report, Visual Studio may sometimes hang idle after launching the debugger. The exact cause of this bug is not known however it appears to be tied to attempting to connect to the development kit. Force closing Visual Studio with the task manager will resolve this issue. However, the problem may persist and varies in frequency.

While premake is not currently supported on Skateboard for PlayStation®, there is a PlayStation®-premake module available [here](https://p.siedev.net/forums/thread/148466/). Note you will require a dev net login. The module allows custom lua scripts to generate project properties targeted for Prospero. This was the method initially used for Skateboard on windows.

* I think it would be nice to bring back support for premake, generating project files through a script is useful and productive. Otherwise, could always create a template in VS and distribute the template across source control.

# Future Work

A lot of work has been implemented into skateboard in the past couple of months, and it is almost ready for game development. One or two imperative systems remain to be implemented, physics, animation, and audio. Animations need finishing off, there is a skeleton loader and blend weights and blend indices parser already in the engine. Additionally, there Is functions to build bind and global poses. These still require testing as I ran out of time to test these systems myself.

Integrating a physics is a due to take place after I leave. As mentioned previously, integrating Nvidia PhysX is recommended as it is cross platform and it is believed to be supported on PlayStation®5 however this will require sending an email to [physxlicensing@nvidia.com](mailto:physxlicensing@nvidia.com) for platform specific files.

A level editor would be nice to have and could allow users to build levels on windows and deploy to various platforms down the line. This would require a couple of additional systems and libraries to ensure level building is a smooth as possible. For instance, gizmos would be a useful implementation for such an endeavour, and imgui gizmos provides a lightweight library developed with ImGui which handles this functionality.

## Material System

In this section I will discuss how I would implement a material system in a game engine or framework which incorporates ENTT and offers users to describe how objects should appear visually in the scene.

### Overview

The material system fundamentally drives the rendering system. In brief, material components are assigned to entities. Each component contains an ID to a material. Simply, materials contain properties which allows the rendering system to render the entity correctly. Additionally, materials will contain parameters which provide the rendering system with further instructions on how the entity should appear visually. Properties can vary depending on the complexity of the system, fundamentally the properties should distinguish between what kind of object is to be rendered, opaque, transparent, translucent, culling, lighting models and blend modes. These are some of the fundamental types of materials as the method in which they are rendered differ significantly from one another.

Using ENTT, the plan going forward would be fetch the material component array and upload data to the GPU. Any material data that has changed since the last frame should be uploaded on the next frame. This is targeted for materials which may contain animated elements, such as scrolling parameters influenced by the application’s elapsed time.

The material component would contain pointers or IDs to shaders stored within the Scene’s data. The shader would contain the instructions to execute while the material buffer contains the data to which alters the geometry’s appearance. This is the plan for the Skateboard at the time of writing this. A basic system such as the one mentioned did exist in the engine’s earlier development, however, many attributes are hard coded and will require de-coupling. Additionally, it is not currently possible to swap out materials on an entity.

A more complicated system beneficial to users with less experience in programming shaders would involve mapping material parameters to instructions. For instance, having pre-written several lighting functions in a shader language, a drop-down menu in the material properties panel would contain several options mapping to the lighting instructions. Some of these options may include, *unlit, lit, toon* and *PBR lit* as some examples. Another example would be the blend mode, and writing the blending in the shader.