Independent Study Report

Victor Shu

Abstraction

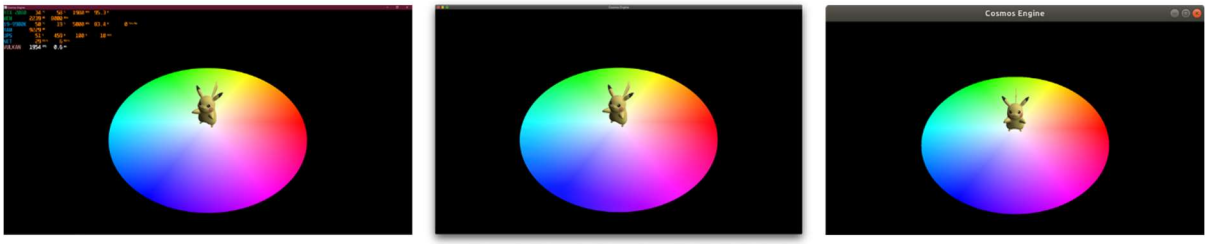
In this report, I'll introduce the composition of the game engine that was made for the course, as well as the usage of the delivered executables.

The final product, Cosmos Engine, is made up of several systems: a rendering system, an input system, a job system, a naïve physics system just for testing the job system, and an audio system. The engine is designed around the idea of making it run on multiple PC operating systems (Windows, Linux, macOS), and the game that made with this engine should run the same on these systems. Thus, the design of the engine will be much different from the one my teammates and I made for IGME.750.01 Game Engine Design and Dev, which only runs on Windows.

There will be three executables in the zip file. One of them is a small application that demonstrates all the systems except for the physics and job system, while the other two demonstrate the difference of the physics system with and without the job system. There will only be executables for Windows, however the source code will compile on all three operating systems.

Overview

The engine is designed around Entity-Component-System. It runs on all three major PC operating systems and have a consistent output.



Cosmos Engine running on Windows, macOS and Ubuntu

The engine is made up of four major systems: a rendering system, an input system, a job system and an audio system.

Build & Run

I use CMake as the build system for the engine. There are two projects: one for the engine and one for the game. The requirements are listed below.

1. General
   - CMake (>= 3.14)
   - Boost (>= 1.69)
   - Vulkan SDK (>= 1.1.108)

- GLFW (= 3.3)
- GLM (= 0.9.9.6)
- Assimp (>= 4.1.0)
- FFmpeg (>= 4.2)
- ShaderConductor (Latest)

2. Windows

There is no additional dependency requirement for Windows. However, it's recommended to install other dependencies with vcpkg (Latest).

3. macOS
   - macOS SDK (Pre-Installed with Xcode)
4. Ubuntu
   - OpenAL (= 1.18.2)

There are other dependencies that are installed by CMake script:

- SPIRV-Cross
- stb collection

Before building the project, compile the ShaderConductor project first. To ensure that CMake finds the binary `ShaderConductorCmd.exe` or `ShaderConductorCmd`, put it along with the dynamic libraries under either of the following location:

- `{CMAKE_BUILD_FOLDER}/ShaderConductor*/Bin`
- `{CMAKE_BUILD_FOLDER}/ShaderConductor/Build/*/Bin`
- `{Environment Variable SHADERCONDUCTOR_ROOT}/Bin`
- `{SHADERCONDUCTOR_PATH}/Bin` (Hint needs to be provided)

Then use CMake to compile the whole project. Make sure that you have internet connection during the process.

After compiling, copy the `Assets` folder at the same location with the compiled binaries. You should see another folder called `Shaders` at the same location.

For all binaries, use mouse to rotate camera, WASD to move the camera, use ESC to exit the game. For TestGame, use mouse left button to play/pause audio, right button to stop audio, use the mouse wheel to zoom the Pikachu model.

Rendering System

The rendering system is designed to be as a middle layer between the engine and different graphics APIs. However, as I used Vulkan as the main API and it's already cross-platform (on macOS, the moltenVK middle layer translates the Vulkan call to Metal) so there's no need to add another graphics API. However, to emphasize, the system is designed to be extendable and can run on multiple graphics APIs.

The system is mainly designed for modern APIs like Vulkan, DirectX 12 and Metal 2, utilizing the new concepts like command buffers/queues. The engine uses shader reflection (for Vulkan, it uses SPIRV-Cross from Khronos Group) to automatically generate the corresponding pipeline and command buffer, which makes it much easier to set the properties of a material.

The rendering system supports all major shading language (compiler provided by the Microsoft project ShaderConductor) so that the user of the engine can use the same shader file for different graphics APIs. Currently in the demo, the rainbow-colored disk is using Vulkan flavored GLSL and the Pikachu model is using HLSL.

Input System

The input system uses GLFW as its backend. GLFW input handles the OS difference very well and it supports a wide range of gamepads. As GLFW does the job, the input system is not designed to have multiple backends for different OS.

The main logic for the input system is the same as the DSEngine from IGME.750.01 Game Engine Design and Dev.

Job System

The job system is based on Molecular Matters' Job System 2.0: Lock-Free Work Stealing. The article uses Win32 API for threading. However, as my implementation is targeting multiple operating systems, I switched to Boost. Libraries like C++ Standard Library and Boost can handle differences between operating systems, compilers and architectures, so I don't need to use the system APIs or compiler extensions myself, but it's critical to understand the memory model and the meaning of the memory order semantics (`relaxed`, `consume`, `acquire`, `release`, `acq_rel`, `seq_cst`) in STL and Boost.

The job system will start (CPU Core – 1) worker threads at the beginning of the initialization. Then, all worker threads will wait for the job queue to be not empty, by using a conditional variable. Each worker threads will try to get a job from its own queue, but when its own queue is empty, it will try to steal a job from others'. Adding a job pushes it at the end of the queue, getting a job pops it from the end of the queue, and stealing a job pops it from the front of the queue. As adding and getting can only happen on the same thread, there will be no race condition

and there's no need to synchronize these two operations. When a race condition happens between pushing and stealing, the stealing operation uses a CAS (compare and swap) operation to test if there is a race condition and will fail if it happens. The popping and stealing will only fight with each other when there is only one job left in the queue. When this happens, the popping operation will try to pop a job from the front of the queue instead of the end to make sure the two operations have the same behavior, then both of the operations will use a CAS operation to test if one of them failed.

The job system uses no locks, only atomic variables.

Audio System

The audio system is the only system where I handle the platform differences myself.

The audio system is made up of two parts: the decoder and the player. The decoder uses FFmpeg, which is a cross-platform library. It is very powerful in terms of decoding. The APIs of FFmpeg are wrapped into a `Decoder` class. The audios are streamed from disk, instead of loading into the memory. One player owns one decoder so that multiple players can play at the same time.

There are different audio APIs on different operating systems. The difference between audio APIs are much more significant than the difference between graphics APIs.

On Windows, XAudio 2 is used. This part is basically copied from DSEngine from IGME.750.01. In XAudio 2, an XAudio 2 engine and a mastering voice will be created for the backend, and each player will create one source voice. The source voices and the mastering voice can have different sample rate and bit depth, and the resampling work is handled by the operating system. The decoder will push raw audio buffers to the source voice. When a buffer is consumed, a function in a callback class will be called. The callback class uses Win32 event to tell the decoder that the buffer queue is full.

On macOS, CoreAudio is used. CoreAudio, like all other major frameworks for Apple operating systems, uses Objective-C. So, for CoreAudio, there is a bridge between Objective-C source file and C++ source files. All CoreAudio components are managed in Objective-C, saved as a C++ pointer, and operations are wrapped into a C++ function. In CoreAudio, a CoreAudio engine should be created first, and a main mixer node can be retrieved from the engine. For each player, a player node will be created and then connected to the main mixer node. The player node and the main mixer node can have different sample rate and bit depth, but the main mixer node requires 32bit float. The decoder will do the format conversion and the operating system will handle the resampling. The decoder will push raw audio buffers to the player node. When a buffer is consumed, a managed Objective-C callback function will be called. This callback function is a wrapper lambda that directly calls a C++ function since C++ function pointers can't be used as Objective-C managed function. The C++ callback function uses a Boost conditional variable to tell the decoder that the buffer queue is full.

On Linux, OpenAL is used. OpenAL is a very old software and it stopped updating years ago, but after researching I found out that it's still commonly used for games. In OpenAL, an OpenAL device and an OpenAL context will be created first. For each player, an OpenAL source will be generated and its ID will be saved. OpenAL doesn't handle the format conversion and resampling, so the decoder will do the work. The decoder will push buffers to the source. OpenAL doesn't provide the callback after a buffer is consumed so every 10ms a thread will poll the status of the source and see if the buffer is consumed.

Results

The following figure shows the performance of the engine with and without the job system. The test platform is Windows 10, the hardware and usage is also shown on the figure. It's obvious that with the job system the engine runs 3 times faster.