Чифир (Chifir) Engine

By Elliot McNeil

Contents

1. Introduction
This document outlines the design of the custom-made 3D game engine for False King and subsequent Randomcode Developers games.

1.1. Language
The engine will be written in C++03, likely using a custom replacement for the STL and/or the C runtime.

1.2. Design
The engine will be based on an entity component system, fairly clean separation between independant components with certain common ones, like platform abstraction, and simple data formats.

1.3. Systems
Each system will be a static or dynamically loaded shared library, and expose a general interface in addition to ECS systems. This will make it easy to add and integrate new features.

2. Engine components

The engine will be made of these pieces:

Component	Components needed	Functionality	Available in tools builds
`Base`	none	containers, basic algorithms, strings, data manipulation and serialization, Unicode handling, startup, shutdown, threading, synchronization, screen output, system information, basic file system functions, input, debugging features	yes
`VideoSystem`	`Base`	abstracts a window or game console screen	yes
`Math`	none	handles higher level math related things than `rtm`	yes
`Utility`	`Base`	localisation, thread pools (maybe), configuration, logging	yes
`Texture`	`Base`	texture format	yes
`Mesh`	`Base`	mesh format	yes
`Pack`	`Base`	package file format	yes
`Launcher`	`Base`, `Utility`	loading an application DLL and the components it needs	yes
`Engine`	`Base`, `Utility`	cameras, scene management, entity component system, commonly used components (for entities), system management	no
`RenderSystem`	`Base`, `Math`, `VideoSystem`	rendering scenes, UIs, anything else	no
`InputSystem`	`Base`	user input	no
`UiSystem`	`Base`, `InputSystem`, `Math`, `RenderSystem`	user interfaces	no
`PhysicsSystem`	`Base`, `Math`	simulates mechanical physics	no
`AnimationSystem`	`Base`, `Math`	controls skeletal animation	no

`AudioSystem`	`Base`, `Math`	handles audio	no
`Game`	`AnimationSystem`, `Base`, `Engine`, `Utility`	game functionality common between client and server, such as prediction and data parsing	no
`GameServer`	`Base`, `Engine`, `PhysicsSystem`, `Utility`	game functionality that happens on the server, such as simulation, player management, etc	no
`GameClient`	`Base`, `Engine`, `InputSystem`, `RenderSystem`, `UiSystem`, `Utility`	game functionality that happens on the client, such as rendering, player input, and possibly prediction	no

3. Platforms

The engine will support at least Windows and Linux. All desktop platforms will use Steam, all others will use the platform's official store.

Platform	Toolchain	Graphics API(s)
Windows	MSVC, GDK	DirectX 12, Vulkan, OpenGL
Linux	LLVM	Vulkan, OpenGL
Xbox Series X|S	MSVC, GDKX	DirectX 12
PlayStation 5	LLVM, PS5 SDK	GNM
Nintendo Switch/Switch 2	LLVM, Switch SDK	Vulkan

These platforms may be supported purely out of personal interest:

Platform	Toolchain	Graphics API(s)	Notes
Xbox 360	Ancient MSVC	DirectX 9	The engine builds, but somehow the XEXs don't have export tables, meaning it doesn't run
PlayStation 3	Ancient GCC, possibly modern LLVM	GCM, OpenGL	Haven't tried this very hard yet, it's probably possible
PlayStation Portable	GCC	OpenGL	Crashes in homebrew startup code, seems like current firmware doesn't support how syscalls are used in it
Bare metal x86	LLVM	Software renderer	This would take a lot of engineering and probably not be worth it

3.1. x86

On x86-based platforms, SIMD is dynamically detected at startup using `cpuid`, so old CPUs still work but modern ones can be used fully. Unfortunately, this does reduce the opportunities for optimization on 32-bit, but that's fine because that isn't going to be the main build that people get.

3.2. Windows

On Windows, the engine does some pretty crazy things. For one, it directly uses `ntdll.dll` instead of `kernel32.dll`, which is mostly a matter of preference. Another thing is that `Base.dll` manually imports system functions from `ntdll.dll` and `user32.dll` and exports them for other modules, in addition to exporting `*_Available` functions. This all lets it run on ancient versions, but then dynamically importing useful functions from newer versions when they're available. The manual importing works by having function pointers and exporting forwarder functions that call them as the real names that `ntdll.dll` or whatever else exports, and then also having functions that check whether the function pointers are null or not. It even avoids having an import table at all by using the PEB to get `ntdll.dll`'s base address, parsing it, and finding `LdrGetProcedureAddress`, and then using that to get other functions normally.

4. Libraries

Library	**Use**
phnt	Exposes internal Windows APIs

5. Engine coding style

The engine is written in C++03 with no STL and no C runtime. This comes with some advantages, but plenty of disadvantages as well.

Pros	Cons
Full control over nearly everything	Have to implement everything
Very portable, everything that can possibly run the engine has a C++03 compiler	Missing really nice things like `constexpr` and range-based for loops, and almost no libraries support C89/C++03
Only the exact functionality needed is implemented	Features not built on existing support code take longer

5.1. Formatting

Just use `clang-format` aggressively. The only thing that I'm pretty sure it doesn't do is adding curly brackets to one-line if statements and loops, which is part of the style. Also, be careful about putting a blank line between headers that shouldn't be sorted alphabetically.

5.2. Types

`public/base/types.h` defines short type names largely based on Rust's type names. Sizes should use the signed `ssize` to make calculation errors easier to see, and the fact that it's a size already gives the indication it can't be negative. Any other type can be unsigned, and for certain things like `operator new()` where using `usize` is required, that's fine too.

5.3. Comments

Comments should explain what code does. At the top of a file that implements something complex, or the main header for a whole component, explain the overall design of the component, any important choices and the reasoning, and whatever limitations exist. Additionally, when functions are complex, add comments explaining what's happening/why it's happening. The memory manager in `base/memory.cpp` is the best example of commenting things so far.

5.4. Naming

Variables are camel case, prefixed with `m_` for private/protected members, `g_` for globals, and `s_` for static globals. Types are Pascal case, prefixed with `C` for classes, `CBase` for abstract classes, and `I` for interfaces, while structs are suffixed with `_t`. Functions are Pascal case, with the name of their component and an underscore as a prefix, like `Base_`. Common abbreviations (like str, len, max, min, alloc, buf, src, dest) are allowed, but obscure ones should be avoided. Try to balance clearness with succintness when naming variables, so they're easier to type.

5.5. Classes vs structs

Classes do things, structs store data. That's the distinction so far.

5.6. Headers

Public headers (ones visible to any component) should include as few headers as possible, and forward declare types where needed. In `.cpp` files, all headers for the things used should be included, not just ones that happen to include the right things. Private headers are more free to include things, and have references to globals inside components, like `base/base.h`.

5.7. Standard library replacement

Because the C runtime and STL aren't used, there are some replacements for the commonly used stuff, and there are also utility functions commonly implemented on top of these, like automatically allocating a buffer for snprintf.

In terms of replacements for the CRT, `base.h` has `Base_Alloc`, `Base_MemSet`, `Base_MemCopy`, and `Base_MemCompare`, and `basicstr.h` has `Base_StrFormat`, `Base_StrCopy`, `Base_StrClone`, and `Base_StrCompare`. They work basically just like `malloc`, `memset`, `memcpy`/`memmove`, `memcmp`, `snprintf`, `strcopy`, `strdup`, and `strcmp`, but because this is still C++, they're overloaded and have behaviour controlled by parameters, which makes them more convenient to use. `Base_MemSet`, `Base_MemCopy`, and `Base_MemCompare` (and the string functions implemented on top of them) also make use of SIMD where possible.

There's not many fancy containers yet, but `CVector<T>` defined in `vector.h` is a working implementation of a dynamic array. Additionally, there's `CLinkedList<T>`, which is used for the free list in the memory allocator, and offers significant user control over the nodes for exactly that reason.

5.8. Assertions and error handling

Assertions are mainly for scenarios that shouldn't happen, and are disabled in retail builds because anything triggering them should be caught in debug/release builds; don't use them for general error handling. For example, if a piece of memory *must* be allocated successfully, like in `operator new()` where the standard technically requires that it not return `nullptr` (even though the standard isn't as relevant for the engine), or an index is outside the valid range, or a parameter is wrong in a way it shouldn't be, then you can use an assert. Normally, you can use the `ASSERT` macro. If a condition isn't the most indicative of why something is wrong, `ASSERT_MSG` lets you add a message. For functions which just succeed or fail, return `false`, `nullptr`, or some other documented value when an error happens. When an unrecoverable error happens, use `Base_Quit` (or `Base_QuitSafe` in functions where avoiding memory allocation is important, such as the memory manager where it would recurse) to kill the engine and show the user an error message.

6. Tools

Tool	Use	Custom made?
FASTBuild	Build system	no
Visual Studio 2022	Code editing, debugging (Windows, consoles)	no
Visual Studio Code/Neovim	Code editing (non-Windows)	no
GDB/LLDB	Debugging (non-Windows)	no
DXC	Shader compiler	no
spirv-cross	Shader converter	no

7. Scene system

Scenes contain entities. Visible entities have components like a mesh or a camera, and transform information. These are some categories of entities:

- Sky/sun/moon entities
- Details like grass, leaves, etc
- Terrain
- Objects like furniture, items, etc
- Players
- NPCs
- Buildings, doors, etc

8. Renderer architecture

The renderer will be implemented in multiple layers, flexible enough to support drawing and post-processing fairly complex scenes, extensible with more techniques and passes, and simple to use.

8.1. Hardware interface

The hardware interface is an abstraction of Vulkan/Direct3D/GNM/whatever other ungodly API I have to deal with. It's low level, and implements render targets, materials, and geometry primitives, as well as special render targets just for going to the screen (they wrap the swap chain images).

- Handles `VkInstance`/`IDXGIFactory`, `VkDevice`/`ID3D12Device`, `VkCommandBuffer`/`ID3D12GraphicsCommandList`, `VkSwapChainKHR`/`IDXGISwapChain`
- Creates and manages geometry (VB+IB), textures, render targets, shaders, materials (texture + shader)
- Handles drawing given geometry + material

8.2. Rendering pipeline

Handles the process of taking data (model, position, etc of objects in scene, and general properties of the world) and using the hardware interface to render and post-process all of it.

- Calls for drawing objects and adding lights
- Uses multiple render passes to light and post-process the scene
- Rasterization-based deferred lighting passes
- Ray-tracing-based lighting passes
- Common post-processing passes

8.3. Render system

Calls into the rendering pipeline to draw scenes from different cameras, such as the player's eyes/over the shoulder, cinematic cameras, mirrors and other reflective surfaces, and literal cameras.

- ECS system that iterates over objects in the scene
- Sets parameters based on scene, such as sky details (even that could be an entity)