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	Some dependencies will need to be ported, and this will be extremely difficult. If LLVM could be modified to target PowerPC Windows, this would become far easier.
PlayStation 3	Ancient GCC, possibly LLVM	GCM, OpenGL	Haven't tried this very hard yet, it's probably possible
PlayStation Portable	GCC	OpenGL	Crashes in homebrew startup code
Bare metal x86	LLVM	Software renderer	This will take a lot of engineering and probably not be worth it

4. Libraries

Library	Use
phnt	Internal Windows APIs
mimalloc	malloc replacement
nvrhi	Graphics API abstraction, makes renderer much easier
rtm	Linear algebra and other math

5. 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

6. Scene system

Scenes contain entities. Entities have components like a “renderable”, which contains a handle to a mesh and other information, and transform information. These are some kinds of entities:

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

7. 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.

7.1. 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

7.2. 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

7.3. 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)