Wizard People Engine TDD

# Build and Project Structure

The project uses up to C++17 features. The tested compilers are VC++ and GCC. The tested platforms are Linux and Windows. The engine uses CMake as its build system and requires at CMake 3.12 or greater.

## CMake

CMake is a meta build system for C/C++ which can generate projects of various other build systems. There are generators like:

* Borland Makefiles
* MSYS Makefiles
* MinGW Makefiles
* NMake Makefiles
* Ninja
* Unix Makefiles
* Most Visual Studio versions
* …

On windows, CMake projects can be developed using Visual Studio’s tooling. By default, Visual Studio uses the Ninja generator. The CMake GUI or CLI can also be used to generate traditional vcxproj and sln files.

The primary CMakeLists.txt file is in the project root. This file mostly includes various other CMake files. One of these is /src/CMakeLists.txt. The configurations for all the sources and libraries of the engine live there.

Many of the project’s dependencies are pulled in from their repositories and compiled along with the engine. This is done with the FetchContent commands.

## Project Organization

### /apps

This will contain the "client code" - the applications which use the game engine. The game executable lives here, along with the demo.

### /art

Various art files that are not assets.

### /cmake

This is for miscellaneous cmake files. Currently, the installer setup is here.

### /cmake\_modules

This folder contains Cmake files for various dependencies.

### /docs

Documentation should go here.

### /include

This is where all the header files go. They are split up by library.

### /src

This is where .cpp files go. The primary engine CMakeLists.txt file is here, which sets up the engine libraries.

### /test

Various tests go here. It is intended for unit test projects, but also contains scenes that test various work-in-progress engine features.

### /thirdparty

This folder contains binaries and sources for various third-party libraries.

## Installer

CPack is used for generating an installer. This uses NSIS and requires it to be installed. And installer can be generated by using a Visual Studio generator to create a solution, then building the PACKAGE project in that solution.

# Engine Modules

## Asset

Asset management code lives here. This includes the asset\_manager, which wraps disk I/O.

## Audio

The audio system, along with the components and loaders live here.

## Collisions

Library for collision BV shapes and intersection tests.

## Core

This contains core engine systems, and rests on top of platform abstractions. The timers, configs, input management, framerate controls and other basic engine functionalities live here.

## ECS

This is the ECS. It is an archetype based ECS, which groups entities by their distinct components sets.

An archetype is a set of components, represented as a std::bitset.

At the lowest level, there are memory\_pool objects that allocate big memory blocks and split them up into chunks that they distribute.

On top of that are archetype\_pool objects, which takes those chunks and initialize entities of different archetypes in them.

chunk\_component\_accessor does the heavy lifting when it comes to getting stuff into and out of the raw memory.

archetype\_pools maintains a map of archetype\_id to archetype\_pool, for each archetype.

state represents all the data in ECS and contains functionality for querying this state. It contains .each<>() functions. These take a set of component types, and create a cache, which has a vector of accessors to entities with the query’s particular set of component types. Each subsequent call to .each<>() can then iterate a contiguous vector of accessors to contiguous blocks of per-entity component data.

world represents and ECS world and contains the data (state) and logic (systems).

## Engine UI

Various IMGUI Engine UIs live here. The imgui\_overlay class contains a list of all the sub-UIs, and handles showing/hiding them.

## Event

The event system lives here.

## Memory

Memory management code that is used by ECS lives here.

## Physics

Physics systems that detect and resolve collisions live here.

## Platform

Platform abstractions live here.

## Rendering

The rendering code of the project. Renderer.cpp is the primary system. No graphics api calls should leak out of this module.

## Transforms

Engine system for updating transform components, and handling transform hierarchies.

## Util

Miscellaneous code.

# Scene file format

The level files are in JSON format, and allow for a basic description of a scene,

with some parenting and data inheritance support.

## Scene

A scene is a collection of entities. Child relationships can be represented as an entity id:

```json

{

"entities": [

{

"entity\_id": 1,

"components": []

},

{

"entity\_id": 2,

"parent\_id": 1,

"components": []

}]

}

```

All the ids in these files should be set, since they ought to be generated by an editor eventually. But I guess there's no real reason not to auto-generate and id if one is omitted.

## Components

Components are identified by the string passed to ecs::register\_component<>();

The component\_loader interface should be implemented by various systems that load and initialize entity components in various ways. These loaders will have the json data from the scene file available to them, along with the ecs entity reference.

Example:

```

"components": [

{

"type": "transform",

"position": [ 1, 2, 3 ],

"rotation": [ 3.14, 3.14, 3.14 ],

"scale": [ 1, 2, 3 ]

},

{

"type": "renderable\_mesh\_static",

"mesh": "this\_isnt\_a\_file.fbx",

"material": "this\_isnt\_a\_file.mat"

}

]

}

```

## Prototypes

A prototype is like a Unity prefab. It is a template for some entity, which can have children entities. Its format is:

```

{

"root": {},

"children": []

}

```

Where the root object and each `children` element are entities. These do not have ids. The elements of "children" have the `root` entity as their implied parent. They can, alternatively, define parents that are within the `children` array by index. So, the property is called `parent\_index`

# Members

Garrett Bennett

Nathan Weise

Nicolas SanJose

Saveliy Baranov

Sylvia Barbero