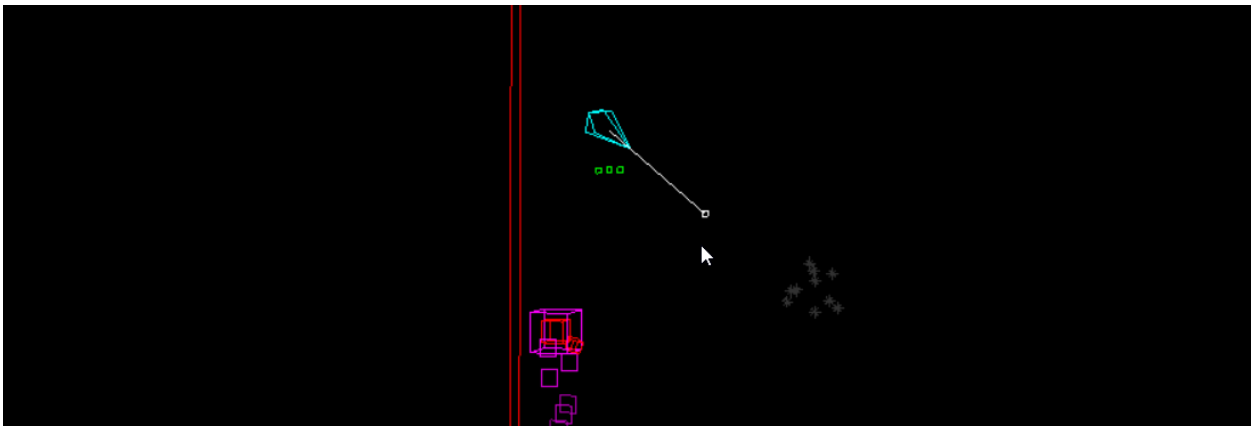
Ubisoft Next Entry



Overview

This document provides some context and descriptions on the systems that power my framework made for UBISOFT NEXT 2024. No external libraries besides standard C++ ones are used, and no direct rendering calls are used either (e.g. OpenGL) as per the specification of the competition.

The framework exposes some useful things that one could expect from a game engine, with the intent on keeping systems extensible and not limiting the user to adhere to a particular set of operations. It is in essence, a collection of utilities onto of an ECS architecture.

Some of the features to expect the framework to provide are:

- 3D renderer
- Particle System
- ECS
- Camera
- Math
- Scenes
- Tests

3D Renderer

The 3D renderer used for this submission is built on top of the NEXT API's line renderer, which uses the OpenGL fixed function pipeline behind the scenes. It does not support shaders or and

native OpenGL calls, but instead operates on **Primitives**, which I define as a collection of Vertices, which are positions.

A primitive can have an arbitrary number of vertices, though 2 should be the minimum to define a line (As I am operating on a line renderer).

Primitives are grouped together into **Meshes,** simply a list of primitives in sequence.

The 3D renderer makes sure to order the faces within it every frame based on the average Z value of the primitives that compose a mesh. Since we lack a depth buffer, this is necessary to simulate depth using the painter's algorithm.

The renderer also operates in similar coordinate spaces as one would expect of traditional 3D techniques, as laid out below:

INPUT: Local space -> World space -> View space -> Clip space -> NDC -> Screen space

The only input needed per functional call for the renderer is a model matrix and a mesh, the view matrix is provided via Camera and the projection matrix is a constant value initialized on startup.

The graphics system also supports clipping, where a primitive is clipped should all of its vertices lay outside the view frustum. The option to render a primitive that should not be clipped (This is useful when you expect to see the edge of a large primitive without any of its vertices on the screen) can be achieved by using *RenderUnclippedMesh.* Otherwise, if you would like to render a mesh that supports clipping with *RenderMesh*.

Particle system

The framework also provides a particle system for use. The particle system takes care of it's own rendering, and only asks the user to provide two calls,

```
void Create(Particle p);
```

- This creates a particle to be rendered.

```
void Render(Graphics& context);
```

- This renders the particle with a given mesh.

Particles are destroyed automatically once their time to live expires, and new particles are freed.

This system uses an object pool which utilizes data locality as well to reduce cache misses and data fragmentation, as we can expect many particles in a scene. This means that a set number

of particles should be defined in the program, and no more can be allocated to that specific particle emitter once defined.

The pool instantiates the number of particles that are defined and leaves them empty at first. When a new particle is created, the pool simply marks down a particle as in use and reinitializes its data for use. Once a particle is deleted, the pool swaps the particle with the unused particle closest to the half of the pool that contains unused particles, effectively keeping all active particles on one half of the pool, which reduces cache misses.

Entity Component System (ECS)

I chose to base this system's architecture as an ECS. The generic object the populates a scene is a GameObject, and entities are expected to derive from a GameObject parent for use in a scene. Components live under the *Component* namespace.

However, this system is a little different than a typical ECS, in the sense that the framework utilizes Entities and Components, but Systems are baked into the entity itself. This was intentional, as it couples together the logic each entity operates on and the entity itself, allowing easier readability and code integration. Reusable logic can be baked into a component as well, as demonstrated in the Transform component's *CreateModelMatrix*.

Components are stored in each GameObject and are unique, meaning no one GameObject can have multiple of the same component. The vector they are stored in is also generic, meaning we can extend the number of components to any arbitrary size without having to change any code.

The component array operates on smart pointers, such that when the game object goes out of scope, all components are also deallocated appropriately.

Camera

Since cameras could vary widely depending on the game or situation, the camera acts more like a container with data with some helper functions in order to create a view matrix for use in the Graphics context.

Math

The framework provides some math classes for use and their supported operations:

- **Vector2f** (Stores 2 floats)

    - Vector subtraction, Vector division, Vector multiplication, Scalar division, scalar multiplication, scalar addition
- **Vector3f** (Stores 3 floats)
    - Vector Addition, Scalar multiplication, Vector Subtraction, Scalar subtraction, Scalar division
- **Vector4f** (Stores 4 floats)
    - Vector Addition
- **Mat4x4** (Stores a 4x4 matrix)
    - Matrix lookup (Parenthesis operator), Matrix Vector Multiplication, Matrix Matrix multiplication, Equivalency between Matrices

The Math package also provides some helpful utility which is used frequently in 3D math, namely:

- Dot product between vectors: *Math::Dot*
- Cross product between vectors: *Math::Cross*
- Normalizing a vector to a unit vector: *Math::Normalize*
- Generate a random float from [min, max]: *Math::RandomFloat*
- Turn degrees into radians: *Math::DegToRad*
- Turn radians into degrees: *Math::RadToDeg*
- Generate a random Boolean value: *Math::RandomBool*
- Find the distance value between two vectors: *Math::Distance*
- Find the magnitude of a vector: *Math::Magnitude*

The MatrixTransform namespace also provides some helpful functions to manipulate matrices:

- Generate a perspective matrix given a FOV, aspect ratio, and near and far planes: *Math::Transform::Perspective*
- Translate a matrix by a Vector3f position: *Math::Transform::Translate*
- Scale a matrix by a Vector3f scale: *Math::Transform::Scale*
- Rotate a matrix in the X axis by some angle in degrees: *Math::Transform::RotateX*
- Rotate a matrix in the Y axis by some angle in degrees: *Math::Transform::RotateY*
- Rotate a matrix in the Z axis by some angle in degrees: *Math::Transform::RotateZ*
- Generate a LookAt matrix given a position, target, and up vector: *Math::Transform::LookAt*
- Generate the inverse of the LookAt matrix given a position, target, and up vector: *Math::Transform::LookAt*

Scenes

The game can be partitioned into scenes using the **Scene** class. A scene class should hold everything related to that scene and is responsible for deallocating everything it allocates. Scenes are then rendered via interface in the **Game.cpp** class, which defines the functions used by the NEXT API. Things are decoupled as much as possible from the Game class and abstracted into Scene instead.

The scene is the context in which all GameObjects exists in and seeks to manage them. Care has been taken to avoid circular dependencies, and the Scene is what runs the systems that require two objects which are unaware of each other to interact.

Tests

Tests are defined in **Tests.cpp** where we can assert and validate aspects of the framework or game, provided that *NEXT_DEBUG* is defined in **Config.h**. This is checked in Game.cpp, and if that define exists, we run all of the tests.