Ubisoft NEXT Programming Documentation

This document will outline the key systems built for **NextGame** and explain how it's used, as well as discuss some of the code architecture behind it.

Table of Contents

Game Engine 1

Hierarchy (Entity Component System) 1

Rendering 2

Mesh Rendering and Vertex Shading 3

Animations (Easing Values) 4

Collision Management 4

Particle System 5

Game Code 6

Game Summary 6

Game Manager 6

Key Game Components 6

Prefabs 7

Assets 8

Incompleted 8

Game Engine

The game systems are designed in an entity component fashion, implementing many of the common features. The primary purpose of the engine is to be able to quickly create game components and relate them together as well as support 3D rendering using the **NextAPI** as a basis.

Hierarchy (Entity Component System)

The structure of ECS designed here will be very familiar with a few alterations to simplify the workflow.

Scene.h/cpp (Singleton)

The scene holds entities and continuously updates and removes them throughout the game loop. Notice the absent of multiple scenes. For this engine, there will only exist a single all-encompassing scene that all **Entities** will live in. For this reason, I've opted to make it a singleton for more convenient global access and enforcement that there will only be one. It also holds a pointer to a single **Camera**, which holds the relevant data for the *View* and *Projection* matrices that the render pipeline will use.

Entity.h/cpp

An entity is essentially an empty container holding only a **Transform** which describes its position/scale/rotation within the game world. Its behavior depends on the attached components.

Component.h/cpp

A component can only exist as part of an entity and gives behavior to said entity. The engine comes with a few important components ranging from collision detection to 3D mesh rendering. The base component class is abstract and is designed to be derived. Writing scripts in this engine depends on writing your own components and allowing the parent entity to call the component each frame.

Rendering

NextAPI offers limited tooling especially when it comes to 3D graphics logic resulting in many new systems designed to support 3D rendering. One of the fundamental requirements for 3D graphics is frequent use of linear algebra in the form of vector and matrix math. To support this, two essential datatypes have been created with operator overloads for ease of use.

- **float3.h/cpp** (3D vector of floats)
- **float4x4 .h/cpp** (4D matrix of floats)

For most 3D dimensional transformations, the **float3** can describe position, scale, and rotation across all three axes. However, due to needing to perform matrix multiplication with a **float4x4** matrix, a hidden fourth component is included in the **float3** for the purpose of matrix transformations. That component is otherwise not used so for simplicity's sake, **float3** should be treated as 3D with their operator overloads reflecting that.

Later, I introduced **Quaternion.h/cpp** as I was experiencing gimble locking when animating rotations. I then replaced the rotation matrix with quaternion rotation in the render pipeline.

Renderable.h/cpp (deriv. Component)

A **Renderable** is an abstract class for a component that has some visual element. It adds a pure virtual **Render()** function to be implemented. Additionally, when a **Renderable** is created or ready to be destroyed, it's immediately adds or remove itself to the **Renderer**.

```
void Renderable::Initialize() {
	Renderer::Get().AddRenderable(this);
}

void Renderable::Destroy() {
	Renderer::Get().RemoveRenderable(this);
}
```

Note that even though they're added to the **Renderer**, the **Entity** that owns the component will already be added to the **Scene**.

Renderer.h/cpp (Singleton)

The **Renderer** holds all the **Renderables** and its job is to run **Render** on them. It does not call **Initialize/Update/Destroy** on them as the **Scene** will take care of running those lifecycle methods.

Graphics Pipeline (MeshFilter)

I will discuss how this is used in the next section, but I want highlight that most of the matrix math for rendering actually happens in the **MeshFilter** component. It holds list of **Triangles** which are just a grouping of 3 float3 points which make up the mesh to render. We then follow the following pipeline.

1. *Apply vertex shading*
2. Apply scaling, rotation, and translation matrices
3. Apply backface culling
4. Apply view matrix with a **Camera**
5. Apply clipping for triangles beyond near and far plane
6. Apply projection onto 2D screen
7. Center and scale to resolution
8. Draw the triangles

There are a couple steps I left out for this engine which include screen-space clipping and depth buffering.

In the case of depth buffering, I did a bit of a hack by adding a way to declare a renderable to always be drawn on top. In my case, I always want the active spaceship to be drawn on top.

```
// renderer.h

void SetTopRenderable(Renderable* _renderable) { topRenderable = _renderable; }
```

When the active spaceship isn't drawn on top, it's really jarring since it's so close to the camera. For objects in the distance, wrong draw order isn't really noticeable. I consider sorting the draws by z-value but that slows things down incredibly so.

Mesh Rendering and Vertex Shading

To render a mesh on screen, you simply add a **MeshFilter** component onto an **Entity** and configure the **Mesh** you want it to use. It will then render the mesh in accordance with the transforms of the parent **Entity**.

```
Entity* entity = Scene::Get().CreateEntity("Test");

MeshFilter * meshFilter = entity->AddComponent<MeshFilter>();
meshFilter->LoadMesh(CubeMesh(1));
meshFilter->SetColor(float3(0.3f, 0.8f, 1.0f));
```

You can pass in any one of the types derived from **Mesh** and it will use that mesh for rendering. Notice you can pass in a value for vertex density of the mesh.

Inspired by GLSL shaders, I felt strongly about implementing vertex shading for more interesting 3D objects and effects. One of the requirements though for vertex shading to be effective is a serviceable number of vertices in a mesh.

For flexibility, I created a few classes that can generate meshes for shape primitives and have variable vertex density depending on how much detail is desired. This could be done by reading off from OBJ files of detailed meshes, but I liked the flexibility to setting mesh detail from within the engine.

With detailed enough meshes. Vertex shading is implemented by running a lambda function across all the vertices prior to any transformations. You can declare the vertex shader by adding it to the **MeshFilter**

```
MeshFilter* meshFilter->SetVertexShader([](float3& vertex) {
	float height = vertex.y;
	height += 5.0f * sinf(2.0f * Time::Get().Elapsed() +
vertex.Distance(float3(0, 0, 0)) * 10.0f
	+ sinf(Time::Get().Elapsed() + vertex.x * vertex.z)
	);
	height = std::round(height);
	vertex.y += height;
	});
```

The above shader will apply a rippling effect on the object, radiating from the center by introducing the elapsed game time.

Animations (Easing Values)

Animations allow you to easily interpolate values of an entity on command. Examples include smoothly panning a camera between targets. To add an animation, hook on an **Animator.cpp/h** component onto an entity. You can then call animate where it will generate an **Animation.cpp.h** that will interpolate the values on each update and remove itself when it's finished.

```
Entity* entity = Scene::Get().CreateEntity("Test");

animator = entity->AddComponent<Animator>();

animator->Animate(entity->GetTransform().position, float3::Zero, 1.0f, new EaseOut);
```

Inside **TimingFunction.cpp.h**, are series of classes for different easing functions. You can subbed whichever into the Animate method as you please. The above example will smoothly translate the entity from its position to (0,0,0) over 1 second.

Collision Management

The **CollisionManager** is fairly simple, it will check collision between every entity in the game world that has the component **Collider**. There are two shapes of colliders, **SphereCollider** and **BoxCollider**. Their sizes can be separately set or more commonly just set to the dimensions of the parent.

```
Entity* entity = Scene::Get().CreateEntity("Test");
BoxCollider* collider = entity->AddComponent<BoxCollider>();
collider->dimensions = entity->GetTransform().scale;
```

For adding behaviors with objects collide, colliders offer a *collisionHook,* which let's you set a function pointer to be called whenever a collision is detected. The system is a bit haphazard as checking for who collides with who is dependent on the way you write your *collisionHook which can be prone to pointer errors*.

```
collider->SetCollisionHook([this](Collider* c1, Collider* c2) {
	if (c2->parentEntity->Name() == "Enemy"") {
		this->TakeDamage()
	}
	if (c2->parentEntity->Name() == "HealthPack") {
		HealthPack* hPack = c2->parentEntity->GetComponent<HealthPack>();
		this->Heal(hpack);
	}
```

Similar to the **Renderables**, and **Collider** that's created will automatically add itself to the **CollisionManager** and will also remove itself when it's done. Currently, the system manually checks collision between every collider regardless of context. Techniques like narrow/broad phase detection or space partitioning would be ideal but is not implemented at this time.

Particle System

Particles are generated strictly from a **ParticleEmitter** component which can be attached any entity and will emit particles from that position. I've included a few parameters that can be adjusted to influence the emission pattern.

```
Entity* entity = Scene::Get().CreateEntity("Test");
ParticleEmitter* emitter = entity->AddComponent<ParticleEmitter>();

emitter->active = true;
emitter->burstSize = 10;
emitter->frequency = 0.05f;
emitter->lifetime = 1.0f;
emitter->size = 20;
emitter->color = color;
emitter->shape = EmissionShape::RADIAL;
emitter->speed = 50.0f;
```

And once again, this function similarly the **Renderables** where **Particles** are created and add themselves to the **ParticleSystem** and when they're lifetime is up, they remove themselves.

One thing to highlight is the rendering for **Particles**. The rendering is very similar to **MeshFilter** but instead of triangles, it is just a line from two points. The rendering pipeline is similar but needed to be adjusted to work with lines such as clipping.

I originally tried rendering shape particles with triangles meshes but saw performance losses with too many on screen. I opted for lines as it let's me render many more at the cost of less interesting particle shapes.

Game Code

Game Summary

Though not code, I've included this in hopes it makes clearer the relationship between the components.

Welcome to Dark Star: Showdown. A 3D, turn based, strategy game that pits you 1v1 against another player amidst a dying star. In the pursuit of advancement, nations have sent legions of ships to the star only to meet their unexpected doom as the star begins dying lightyears ahead of schedule. You and another player pilot the remaining ships from opposing nations. As death draws near, the two of you face off in a final showdown of triumph.

The game starts with both players on opposite sides of the star. The goal is to confront the other player and shoot them down before the star implodes and kills everyone. Both ships start rather weak, but they can destroy surrounding asteroids to gather resources and power themselves up. As the star weakens, it will occasionally pulse, pulling the players closer together. Eventually the players will be forced to either fight or wait for both their dooms.

Game Manager

This is an entity that manages the flow of the game from switching between camera views to switching between UI's depending on the state of the game. It's comprised of to components that talk closely to each other.

- **ViewManager.cpp.h**
- **MenuUI.cpp.h**

The more important is **ViewManager** which essentially decides what should be on screen. It implements the **Observer** pattern and acts as an observer into the other key game objects. It's able to determine game state by listening for events from **ObservableComponents**.

```
void ViewManager::OnNotify(const Entity* entity, GameEvent event) {
	switch (event) {
	case PLAYERA_WIN:
		break;
	case PLAYERB_WIN:
		break;
	case STAR_PULSE:
		break;
	case SUPERNOVA:
		break;
	default:
		break;
	}
}
```

Key Game Components

DarkStar

The great thematic star for this game. It sets up all its required components on initialize like collider's and particle emitters. In summary, it has the following behavior.

- On collisions, it will *pulse()* resulting in an **Animation** trigger causing it shrink and pulsate with its vertex shader.
- It is an **ObservableComponent** and notifies events like when it pulses so the GameManager knows when to switch views and refund fuel/apply damage to the ships.
- Generates a whole field of **Asteroids** around the star uniformly randomly
    - **Asteroids** are another entity with colliders and shaders an such
    - The mesh is randomly generated using the vertex shader to create a unique, bumpy rock
    - When **Asteroids** are destroyed, they explode into **Scrap**
- **Scrap** is an entity that can be collected by ships in radius

Ship

The ships that each of the player's controls. In its initialize it sets up all the needed components to the parent such as colliders, particle emitters, etc. In summary it has the following behavior

- Multi-axis movement
- Fire's **Bullet's** that can collide with other entities
- Automatically adjusts the scene camera to a third person view on the ship
- Has an *active* variable the will release camera and input control when *false*
- Is an **ObservableComponent** and notifies events like losing

I create two of these, one for each player and place them on either side of the **DarkStar**

Prefabs

I consider doing something more data-driven but I ended up shying away from having to file parse. Instead, I opted for a set of macros that creates a reuseable function for creating entities and attaching components to them.

Using it just requires defining a prefab in **Prefabs.h** and then implementing in **Prefabs.cpp.**

```
// Prefabs.h

DEFINE_PREFAB(Wall);

// Prefabs.cpp

IMPLEMENT_PREFAB(Wall, {
	MeshFilter * meshFilter = entity->AddComponent<MeshFilter>();
	meshFilter->LoadMesh(CubeMesh(1));
	meshFilter->SetColor(float3(0.3f, 0.8f, 1.0f));
	BoxCollider* collider = entity->AddComponent<BoxCollider>();
	collider->dimensions = offset.scale;
	});

// NextGame.mcpp

Prefabs::Wall(Transform());
```

Notice, you can also pass a transform to easily set the position of the prefab.

I didn't use this as much as I intended as I ended up having components add components to their parents on initialize instead. Still, it came in handy for testing and quickly placing objects around the world.

Assets

I acquired some free use audio assets to use within the game.

https://tallbeard.itch.io/three-red-hearts-prepare-to-dev

https://horror-studio.itch.io/8bit-sfx-pack-100-pack

Incompleted

- InputManager to handle keydown and keyup states. In the meantime I did some boolean hacks to get it working but looking forward, a dedicated system would be ideal.
- Object pooling to help lower memory demand with frequently instantiated objects like bullets or particle effects.
- More mesh shapes (a better one for the space ship, maybe a cone)
- Skybox (something like stars or galaxies)
- UI for upgrade stats