NextCore User Docs

Welcome to the NextCore user docs! Inside you'll find a brief outline of the different systems and features of NextCore, as well as some example code to get you started. Both the front-end and back-end are documented, so feel free to dive in wherever you want to contribute!

Below is the list of NextCore's major features, from user-facing features you'll use as a client of NextCore, to more internal systems and processes that are great to be conscious of while using NextCore.

Table of Contents

Major Client features

Entity Component System

Back to Top
While NextCore doesn't strictly feature an entity component system with true separation between data and behaviour, NextCore takes a more object-oriented approach to the ECS problem. NextCore makes heavy use of composition in its design, akin to an ECS where the components and the systems are effectively the same thing. So NextCore uses more of an EC.

In NextCore, Users author **Components** that interact with the game world. Components can be thought of as individual behaviours that contribute to one entity, or if users so choose, pure data classes.

Components don't exist on their own, though. Every component is owned by an **Entity**. Entities can be thought of as a container of components. Users can freely pass around entities, and use them as handles to retrieve components.

As for components, take this example of a first person free-camera implementation with gamepad controls. We will refer back to this example to illustrate the other major features of NextCore as we move through them.

Note: In this documentation, we will be referring to all "Behaviours" as "Components". Functionally, there is no different between these two at the moment, but it is recommended that users inherit all client objects from Behaviour to ease the transition later if new functionality specific to behaviours is added.

Skip the code

```
// SimpleFPSCamera.h
#include <HeaderInclude.h>

class SimpleFpsCamera : public Next::Behaviour
{
    ReflectDeclare(SimpleFpsCamera, Next::Behaviour)
```

```
public:
    void OnCreate() override;
    void OnUpdate() override;

private:
    float m_moveSpeed = 5;
    float m_turnSpeed = 180;

    Next::Transform* m_transform;

    ReflectMembers(
        ReflectField(m_moveSpeed, r_name = "Move Speed")
        ReflectField(m_turnSpeed, r_description = "How fast the camera turns per second in degrees")
    )
}

// SimpleFPSCamera.cpp
#include "SimpleFPSCamera.h"

#include <MinimalInclude.h>

// Ensure SimpleFpsCamera is registered with the reflection system.
// Can also be called explicitly in init code with
// Next::Reflection::Type::Register<SimpleFpsCamera>();
ReflectRegister(SimpleFpsCamera)

using namespace Next;

void SimpleFpsCamera::OnCreate()
{
    m_transform = Transform();

    // Camera is fire and forget
    auto* camera = AddComponent<Camera>();
    camera->SetFov(105, FovType::Horizontal);
}

void SimpleFpsCamera::OnUpdate()
{
    Vector3 position = m_transform->GetPosition();
    Vector3 rotation = m_transform->GetRotation();

    position += m_transform->Forward() * Input::GetAxis(Axis::Vertical)     * m_moveSpeed * Time::DeltaTime();
    position += m_transform->Right()   * Input::GetAxis(Axis::Horizontal)   * m_moveSpeed * Time::DeltaTime();
    position += m_transform->Up()      * Input::GetAxis(Axis::RightTrigger) * m_moveSpeed * Time::DeltaTime();
    position -= m_transform->Up()      * Input::GetAxis(Axis::LeftTrigger)  * m_moveSpeed * Time::DeltaTime();

    rotation.x += Input::GetAxis(Axis::VerticalLook)   * m_turnSpeed * Time::DeltaTime();
    rotation.y += Input::GetAxis(Axis::HorizontalLook) * m_turnSpeed * Time::DeltaTime();
    rotation.x = std::clamp(rotation.x, -85.f, 85.f);

    m_transform->SetPosition(position);
    m_transform->SetRotation(rotation);
}
```

Components in NextCore can override a number of event functions that are called at certain points in the game loop. See `NextCore/Scripting/Component.h` (region Event Functions) for more information on the event functions that can be overridden.

For example, `OnCreate` is called immediately after the component is constructed in the registry, and `OnUpdate` is called once per frame. Users can also manipulate the components of entities with the `AddComponent`, `GetComponent`, and `RemoveComponent` functions. Components can be accessed through both templates and reflection. For more information on reflection, see the section on Reflection.

```
// Templates
AddComponent<SimpleFpsCamera>();

// Reflection
namespace Reflection = Next::Reflection;

AddComponent(SimpleFpsCamera::GetStaticType());
AddComponent(Reflection::GetStaticId<SimpleFpsCamera>());
AddComponent(Reflection::Type::Get<SimpleFpsCamera>());
```

Note: Due to how components are handled internally, function-local pointers are not guaranteed to be valid after calls to `AddComponent` and `RemoveComponent`. Ensure that when using function-local pointers to components, `GetComponent` is used after every call to Add and Remove. This rule **does not** apply to pointers that are member-fields of the component. For more information, see the section on Reference Tracking.

3D Renderer

Back to Top

Reflection

Back to Top
One of the pillars of NextCore is code introspection, or reflection (Read more). In practice, this allows users to:

- View type information at run time, like the members of a class and their types, offsets, sizes, etc, and get and set their
  (*member functions and statics currently not supported*)
- Allocate and de-allocate instances of types either on the heap or in arbitrary locations (query for the size of the buffer with `GetSize()`)
- Set descriptive information at runtime like a display name and a description, and access it at runtime (Useful for writing automatically populating UI like a details windows)

Every type that is to be used with the reflection type must be registered. This can be done by either using **one** instance of `ReflectRegister()` per type in a translation unit at global scope (Recommended to place in the translation unit for that type, ie place the register declaration for SimpleFpsCamera in SimpleFpsCamera.cpp), or by explicitly calling `Reflection::Type::Register<T>()` in some initialization code, or before you need to pass around the type / get access to it.
Note that calling `Reflection::Type::Get<T>()` internally calls `Type::Register<T>()` for you, but calls to functions that take in a type or a type id will fail if the type hasn't already been registered.

It's upon this feature the entire entity component system of NextCore is built, and so reflection is tightly woven into this system. You can do all of the things that you can do with template arguments, with instances of `Type` and `TypeId` (Example). internally, virtually no static typing is used, so you can feel free to use static or dynamic typing as you see fit.

Reference Tracking

Back to Top

One of the most exciting features in NextCore is what we call reference tracking. Reference tracking is akin to using smart pointers and reference counting, but with a much simpler user API. In fact, reference tracking isn't opt-in as it's baked into the component system, and users don't need to do anything special to make use of the system.

Reference tracking in NextCore keeps track of all of the component-pointer members of all components, and automatically updates the values of the pointers when the reference is invalidated by resizing or otherwise the component being moved around in memory.

Take the SimpleFpsCamera for example. In this case, we're taking a look at the `m_transform` field. Whenever a component is created, there's a chance that the existing pool allocator wont have enough room for the component, and so it will have to resize. Thanks for reference tracking, users can reference other components through raw pointers, and the reference tracking system will automatically update the pointer value for the user, allowing the user to focus on the high-level details.

Major Backend Systems

Entity Registry and Component Pools

Renderer

Tests