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!

Table of Contents

Major features

Back to Top

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.

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.

```
// 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)
        ReflectField(m_turnSpeed)
    )
}

// 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` (line 65) 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

Reference Tracking

Back to Top