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
  - 3D Renderer
  - Reflection
  - Reference Tracking
  - Scene System
- Major Backend Systems
  - Entity Registry and Component Pools
  - Tests

Major Client features

Entity Component System

Summary

NextCore has an "Entity component system" based on composition and object oriented classes. Users author components that add behaviours, that are then attached to entities in the game world.

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
{
    // Setup member functions and other things for the reflection system
    ComponentDeclare(SimpleFpsCamera, Next::Behaviour)

    // ReflectDeclare is used for all non-component classes, namely scenes
    // 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_transform) // Needed for reference tracking to work
        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

Summary

NextCore is a 3D engine, allowing users to import 3D textured models and move them around in 3D space, as well as a scene graph for relative movement between entities.

Back to Top

NextCore comes with a built in software 3D renderer. This means that most of the work you'll be doing in NextCore will be in 3D! At the user level, users will create `ModelRenderers`, and populate its `model` field with a loaded in model by calling the `Model::Create("filename")` function. Currently, only .obj files are supported (Please ensure vertex positions and texture positions are exported!). Both quads and triangle based models are supported. NextCore will also automatically load in the texture file with the same name in the same directory as the file (must be .bmp and have equal width and height). The model will then be rendered based on its entity's transform (position, rotation, scale).

NextCore features a scene graph, that is entities have parents that will affect their world-space position. You can modify both global and local position, rotation, and scale of any entity. At the user level, positions and scale are 3D vectors, and rotations are represented with 3x3 matrices.

NextCore also features simple lighting, with directional lights and point lights. You can use these features buy creating `Light` components and modifying their values. Directional lights use their transform's forward direction to determine light direction, and point lights use their transform's position to determine light position.

Reflection

Summary

NextCore features code introspection based on C#'s api. This allows users to query a type's data members and arbitrarily construct objects at runtime, as well as support some of the engine's more interesting features, like reference tracking.

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)

See `NextCore/Reflection/Type.h`, `NextCore/Reflection/Reflection.h` for more info.

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

Summary

Inside components, raw member pointers to other components, so long as they are registered with the reflection system, will automatically be changed when the component moves around in memory, meaning users only need to check if the pointer is null to check for reference invalidation.

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.

Scene System

Summary

NextCore is built on scenes. Scenes control the initialization of batches of entities that can be reused multiple times. For example, users can have a menu, gameplay, and credits scene with different sets of components that can be moved between at will.

Back to Top

NextCore features a scene system that allows for construction and destruction of scenes easily at runtime. Scenes are a necessary part of using NextCore, as they are used in place of calling initialization functions. As a user, you can inherit a class from `Next::Scene` (and register it with the reflection system with `ReflectRegister`!). It has two virtual functions, one for creation and one for destruction. Generally you will only be using the creation function, as this is where you set up the scene by creating all of the entities for the scene (like a player character, camera, etc).

To switch to and from scenes, use the `SceneManager::LoadScene` function. This function takes in the fully qualified name of the scene class, its reflection type or typeid, or the scene itself as a template parameter. So long as the scene is registered with the reflection system, it will have automatically be registered with the SceneManager as well.

To tell NextCore what scene you want the game to start in, simply place `StartingScene(<YourSceneType>)` in **one** cpp file (We recommend a config.cpp file in the root directory of the game module). After you use this macro, NextCore does the rest.

Major Backend Systems

Entity Registry and Component Pools

Summary

NextCore uses pool allocators for memory management, and maintains type-agnostic information about all of the different component pools so that the entity registry can manage what entities own what components without knowing their types at compile time.

Back to Top

Under the hood, instances of the `Entity` class are just a wrapper for an opaque `EntityId` that is a uint32_t. The entity class allows users to interface with the entity registry, which contains information about every active entity and each component pool. The registry itself is uninteresting, as it is also just an interface for the component pools.

NextCore stores it's components in component pools. Component pools are effectively pool allocators, in that a chunk of memory is allocated at start, and a list of indices are generated that will be used to index into the array. When a component is requested for creation, the next available index is popped from the queue, and the requested component is constructed in place at that position in the queue using its corresponding TypedFactory (Utilizes polymorphism to in-place construct objects when you know the size at runtime but not the type). The index is then stored as the value in a key value pair alongside the entity that owns the component for easy retrieval. Note: Each entity can own only one component of a given type.

When a component pool is resized, of course it needs to be reallocated to keep all of the components contiguous in memory. To not break any pointers in components to other components, reference tracking is used. Internally, when a component pool with type T is about to be resized, the registry looks through eery other component (including type T) that has a field of type pointer to T. The entity id is cached alongside the raw value of the pointer. Once the resize is completed and the arrays are (likely) no longer in the same location in memory, the registry goes back through and reconstructs the pointers using the cached value and entity id. This system is at the heart of the component pool system, and is how we got away with not using smart pointers or otherwise reference counting.

Tests

NextCore has a number of unit tests aimed at squashing bugs in key portions of the program. Currently, proper tests exist for the entity registry and components system, reference tracking, and reflection, as well as some preliminary tests for math functions. See `Tests/` for more info.