Components

An overview of Components

Components are a special type of **Object** that **Actors** can attach to themselves as sub-objects. Components are useful for sharing common behaviors, such as the ability to display a visual representation, play sounds. They can also represent project-specific concepts, such as the way a vehicle interprets input and changes its own velocity and orientation. For example, a project with user-controllable cars, aircraft, and boats could implement the differences in vehicle control and movement by changing which Component a vehicle Actor uses.

Actor Components

`UActorComponent` is the base class for all Components. Since Components are the only way to render meshes and images, implement collision, and play audio, everything the player sees or interacts with in the world when playing the game is ultimately the work of some type of Component.

There are a few major classes to understand when creating your own Components: **Actor Components**, **Scene Components**, and **Primitive Components**.

- **Actor Components** (class `UActorComponent`) are most useful for abstract behaviors such as movement, inventory or attribute management, and other non-physical concepts.

Actor Components do not have a transform, meaning they do not have any physical location or rotation in the world.

- **Scene Components** (class `USceneComponent`, a child of `UActorComponent`) support location-based behaviors that do not require a geometric representation. This includes spring arms, cameras, physical forces and constraints (but not physical objects), and even audio.
- **Primitive Components** (class `UPrimitiveComponent`, a child of `USceneComponent`) are Scene Components with geometric representation, which is generally used to render visual elements or to collide or overlap with physical objects. This includes Static or skeletal meshes, sprites or billboards, and particle systems as well as box, capsule, and sphere collision volumes.

Registering Components

In order for Actor Components to update each frame and affect the scene, the Engine must **register** them. This happens automatically for Components created as sub-objects of an Actor during that Actor's spawning process. However, manual registration is available for Components created during play. The `RegisterComponent` function provides this functionality, with the requirement that the Component is associated with an Actor.

> Registering a Component during play can impact performance, so you should do so only in cases where it is necessary.

Register Events

In the process of registering a Component, the Engine associates the Component with the scene, making it available for per-frame updates, as well as running the following `UActorComponent` functions:

Function	Description
`OnRegister`	This function can be overridden to add code when registering a Component.

Function	Description
`CreateRenderState`	Initializes the render state for the Component.
`OnCreatePhysicsState`	Initializes the physics state for the Component.

Unregistering Components

To remove Actor Components from update, simulation, or rendering processes, you can unregister it with the `UnregisterComponent` function.

Unregister Events

The `UActorComponent` functions below run when a Component unregisters.

Function	Description
`OnUnregister`	This function can be overridden to add code when unregistering a Component.
`DestroyRenderState`	Uninitializes the render state for the Component.
`OnDestroyPhysicsState`	Uninitializes the physics state of the Component.

Updating

Actor Components have the ability to update each frame in a manner similar to Actors. The `TickComponent` function enables Components to run code on each frame. For example, **USkeletalMeshComponent** uses its `TickComponent` function to update animations and skeletal controllers, while **UParticleSystemComponent** updates its emitters and handles particle events.

By default, Actor Components do not update. In order to make your Actor Component update each frame, you must enable ticking by setting `PrimaryComponentTick.bCanEverTick` to

`true` in its constructor. After that, either in the constructor or elsewhere, you must call `PrimaryComponentTick.SetTickFunctionEnable(true)` to turn updates on. You can later deactivate ticking by calling `PrimaryComponentTick.SetTickFunctionEnable(false)`. If you know your Component will never need updates, or if you intend to call your own update function manually (perhaps from an owning Actor class), you can simply leave `PrimaryComponentTick.bCanEverTick` with its default `false` and get a slight performance boost.

Render State

In order to render, an Actor Component must create a render state. This render state also informs the Engine when something has changed about the Component that requires its render data to be updated. When such a change occurs, the render state is marked "dirty". If you build your own Components, you can mark render data as dirty with the `MarkRenderStateDirty` function. At the end of the frame, all dirty Components have their render data updated in the Engine. Scene Components (including Primitive Components) create render states by default, while Actor Components do not.

Physics State

To interact with the Engine's physics simulation system, an Actor Component needs a physics state. Physics states update immediately when changes occur, preventing issues like "frame-behind" artifacts and removing the need for a "dirty" marker. By default, Actor Components and Scene Components do not have physics states, but Primitive Components do. Override the `ShouldCreatePhysicsState` function to determine whether or not instances of your Component class need physics states.

If your class uses physics, simply returning `true` is not advised. See the `UPrimitiveComponent` version of the function to get an idea of some of the situations in which you should not create a physics state, such as during the Component's destruction. You can also return `Super::ShouldCreatePhysicsState` in cases where you would normally return `true`.

Visualization Components

Some Actors and Components have no visual representation, making them difficult to select, or have important properties that are not visible. Developers can add extra Components to display information while working in the Editor, but these extra Components are not needed during Play In Editor, or when running a packaged build. To address this, the Editor supports the concept of **Visualization Components**, which are ordinary Components that exist only when working in the Editor.

To make a Visualization Component, create any regular Component and call `SetIsVisualizationComponent` on it. Since the Component does not need to exist outside of the Editor, all references to it should be inside of preprocessor checks against `WITH_EDITORONLY_DATA` or `WITH_EDITOR`. This will ensure that packaged builds are unaffected by these Components and are guaranteed not to reference them anywhere in code. As an example, the **Camera Component** uses several other Components to display helpful information in the Editor, including a **Draw Frustum Component** to show its view frustum. In the header file, the Draw Frustum Component is defined within the class as follows:

```
#if WITH_EDITORONLY_DATA
// The frustum component used to show visually where the camera field of view is
class UDrawFrustumComponent* DrawFrustum;
// ...
#endif

```

Copy full snippet

Similarly, all references to this this Component will be inside of preprocessor checks against `WITH_EDITORONLY_DATA` in the source file. This code, inside of a `WITH_EDITORONLY_DATA` check within `OnRegister`, checks to see if the Camera Component is attached to a valid Actor, and then adds the Draw Frustum Component code:

```
void UCameraComponent::OnRegister()
{
#if WITH_EDITORONLY_DATA
if (AActor* MyOwner = GetOwner())
{
```

```
// ...
if (DrawFrustum == nullptr)
{
DrawFrustum = NewObject<UDrawFrustumComponent>(MyOwner, NAME_None,
RF_Transactional | RF_TextExportTransient);
DrawFrustum->SetupAttachment(this);
DrawFrustum->SetIsVisualizationComponent(true);
// ...
}
}
// ...
#endif
Super::OnRegister();
// ... Additional code (to run in all builds) goes here ...
}

```

Copy full snippet

`DrawFrustum` now exists only in the Editor and is considered a Visualization Component, meaning that it won't appear during in-Editor playtesting.

Scene Components

A Scene Component is an Actor Component that exists at a specific physical position in the world. This position is defined by a **transform** (class `FTransform`), containing the location, rotation, and scale of the Component. Scene Components have the ability to form trees by attaching to each other, and Actors can designate a single Scene Component as "root", meaning that the Actor's world location, rotation, and scale are drawn from that Component.

Attachment

Only Scene Components (`USceneComponent` and its child classes) can attach to one another, due to the requirement for transforms to describe the spatial relationship between the child and parent Components. While a Scene Component can have any number of children, it can have only one parent, or can be placed directly in the world. The Scene Component system does not support attachment cycles. The two primary methods are `SetupAttachment`, which

is useful in constructors and when dealing with Components that haven't been registered yet, and `AttachToComponent`, which attaches one Scene Component to another immediately and is useful during play. This attachment system also enables attaching Actors to each other by attaching the root Component of one Actor to a Component belonging to another.

Primitive Components

Primitive Components (class `UPrimitiveComponent`) are Scene Components that contain or generate some sort of geometry, generally for rendering or collision purposes. There are several subclasses for the various types of geometry, but the most common by far are the **Box Component**, **Capsule Component**, **Static Mesh Component**, and **Skeletal Mesh Component**. Box Component and Capsule Component generate invisible geometry for collision detection, while Static Mesh Component and Skeletal Mesh Component contain pre-built geometry that is rendered, and can also be used for collision detection if desired.

Scene Proxy

The **Scene Proxy** (class `FPrimitiveSceneProxy`) of a Primitive Component encapsulates scene data that the Engine uses to render the Component in parallel to the game thread. Each type of primitive has its own Scene Proxy child class to hold the specific render data it needs.

See Rendering System Overview for more details on primitives and rendering geometry.