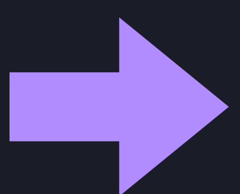
# UNITY2UNREAL

1. LIFECYCLE FUNCTIONS
2. Update vs Tick
3. TRANSFORM
4. PHYSICS FORCES & RIGIDBODIES
5. COROUTINES vs TIMERS
6. SERIALIZATION & DATA PERSISTENCE



# UNITY2UNREAL

## PART 1: LIFECYCLE FUNCTIONS

### Awake() & Start()

**Awake()** is called when the script instance is being loaded, ideal for initial setup. **Start()** is called before the first frame update, only if the script instance is enabled. Unity calls all **Awake()** methods in the scene before any **Start()** methods.

```
void Awake() {
    // Initialize variables here
}

void Start() {
    // Setup dependencies, after Awake
}
```

### OnDestroy()

Called when the object is being destroyed. Use for cleanup. Also triggers on domain reload or scene unload in the Editor.

```
void OnDestroy() {
    // Cleanup resources
}
```

Unreal's lifecycle is often more explicit, especially in C++. **BeginPlay** is the common go-to like Unity's **Start**.

### PostInitializeComponents() & BeginPlay()

**PostInitializeComponents()** is called after all components are initialized *(C++ only; Blueprints don't expose this)*. **BeginPlay()** is called when the Actor enters the game world, similar to Unity's Start. For Blueprint-only workflows, put 'Awake-style' logic in the *Construction Script* or the very start of **BeginPlay()**.

```
virtual void PostInitializeComponents() override;
virtual void BeginPlay() override;

// In .cpp
void AMyActor::PostInitializeComponents() {
    Super::PostInitializeComponents();
    // Setup after components are initialized
}

void AMyActor::BeginPlay() {
    Super::BeginPlay();
    // Actor is ready
}
```

### EndPlay()

Called when an actor is being removed from the game. Similar to Unity's **OnDestroy**. Runs when the actor is destroyed or the level/PIE session ends; fires before the final **Destroyed()** cleanup.

```
virtual void EndPlay(const EEndPlayReason::Type
EndPlayReason) override;

// In .cpp
void AMyActor::EndPlay(const EEndPlayReason::Type
EndPlayReason) {
    Super::EndPlay(EndPlayReason);
    // Cleanup resources
}
```

# UNITY2UNREAL

## PART 2: UPDATE vs TICK

### Unity: Opt-out

**Default**: All active *MonoBehaviour* scripts automatically receive update callbacks if they declare the method.

**Control**: Via *MonoBehaviour.enabled* or GameObject active state.

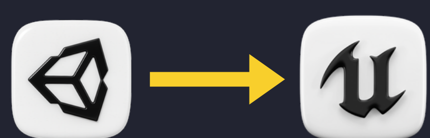
### Unreal Engine: Opt-in

**Default**: Nothing ticks by default. Actors/Components must explicitly enable ticking and register Tick().

**Control**: Via *bCanEverTick*, *TickGroup*, *TickInterval*, and *AddTickPrerequisiteActor*.

| Purpose | Unity Callback | Unreal Equivalent(s) |
|---|---|---|
| Standard per-frame logic | *void Update()* | *virtual void Tick(float DeltaSeconds)* (default) |
| Physics-step logic | *void FixedUpdate()* | *Tick()* in *TG_PrePhysics* or *TG_PostPhysics* |
| Post-update adjustments (camera, IK) | *void LateUpdate()* | *Tick()* in *TG_PostPhysics* or *TG_PostUpdateWork* |
| Low-frequency / background work | *InvokeRepeating*, *coroutines*, custom timers | Set *TickInterval* on actor/component's *PrimaryActorTick* |

## Porting from Unity to Unreal



- **Physics dependency:** Unity's *FixedUpdate* often maps to UE's *TG_PrePhysics* (for physics input) or *TG_PostPhysics* (for physics output).
- **Replace *LateUpdate*:** Use UE's later tick groups (e.g., *TG_PostUpdateWork*) for camera follow or post-processing logic.
- Leverage ***TickInterval***: Many Unity scripts using *InvokeRepeating* can efficiently use Unreal's per-object TickInterval.
- **Exploit prerequisites:** Unreal allows explicit ordering between dependent actors using prerequisites, offering more control than Unity's Script Execution Order.

# UNITY2UNREAL

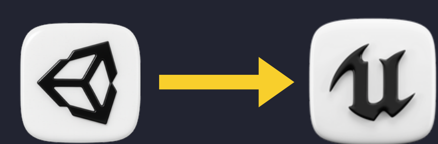
## PART 3: TRANSFORM

| UNITY | UNREAL ENGINE |
|---|---|
| Object-style API: Uses direct properties on Transform (e.g. transform.position) | Function-style API: Uses SetActorLocation(), GetActorRotation(), etc. |
| Rotation type: Quaternion with optional Euler helpers | FRotator (Yaw, Pitch, Roll) |
| Physics integration: Not built into Transform | Supports collision via bSweep, Teleport flags in movement functions |

## Conceptual Mapping (Unity ↔ Unreal)

| Concept | Unity | Nearest Unreal Equivalent |
|---|---|---|
| **World position getter** | `transform.position` | `GetActorLocation()` |
| **World position setter** | `transform.position = ... or Translate()` | `SetActorLocation() with bSweep & Teleport flags` |
| **Local position setter** | `transform.localPosition = ...` | `SetActorRelativeLocation() (if Actor is attached, or per-component)` |
| **World rotation getter/setter** | `transform.rotation (Quaternion)` | `Get/SetActorRotation() (FRotator)` |
| **Local rotation** | `transform.localRotation` | `SetActorRelativeRotation()` |
| **Scale (world vs. relative)** | `transform.localScale (relative) / lossyScale (computed world)` | `SetActorScale3D() (world) / SetActorRelativeScale3D()` |
| **Additive offset** | `Translate, RotateAround` | `AddActorWorldOffset/Rotation (+ local variants)` |
| **Convert point between spaces** | `TransformPoint, InverseTransformPoint` | `USceneComponent::K2_GetComponentToWorld() + manual FTransform::TransformPosition` |

## Migration Tip: Unity vs Unreal



```
transform.Translate(Vector3.forward *
speed * Time.deltaTime);
```

```
AddActorLocalOffset(GetActorForwardVector() * Speed *
DeltaTime, /*bSweep=*/false);
```

# UNITY2UNREAL

## PART 4: PHYSICS FORCES & RIGIDBODIES

**Unity – "Component = Physics"**

Adding a Rigidbody component immediately puts the GameObject under PhysX simulation—gravity on, collisions active, and integrated every FixedUpdate frame.

**Unreal – "Flag = Physics"**

Static or skeletal meshes don't simulate until you enable it. Tick Simulate Physics in the Details panel or call UPrimitiveComponent::SetSimulatePhysics at runtime.

## Conceptual Mapping (Unity ↔ Unreal)

| Concept | Unity | Nearest Unreal Equivalent |
|---|---|---|
| Enable simulation | `Rigidbody (isKinematic)` | `UPrimitiveComponent::SetSimulatePhysics` |
| Continuous force | `Rigidbody.AddForce(Vector3, ForceMode.Force)` | `UPrimitiveComponent::AddForce(FVector)` |
| Instant impulse | `Rigidbody.AddForce(..., ForceMode.Impulse)` | `UPrimitiveComponent::AddImpulse` |
| Apply torque | `Rigidbody.AddTorque(Vector3)` | `UPrimitiveComponent::AddTorqueInRadians` |
| Drag / damping | `Rigidbody.drag, Rigidbody.angularDrag` | `UPrimitiveComponent::SetLinearDamping, SetAngularDamping` |
| Toggle gravity | `Rigidbody.useGravity` | `UPrimitiveComponent::SetEnableGravity` |

## CODE SNIPPETS

```
// Unity (impulse shot)
Rigidbody rb = GetComponent<Rigidbody>();
rb.AddForce(Vector3.forward * 500f, ForceMode.Impulse);
```

```
// Unreal (C++)
UStaticMeshComponent* MeshComp = FindComponentByClass<UStaticMeshComponent>();
MeshComp->SetSimulatePhysics(true);
MeshComp->AddImpulse(FVector(500.f, 0.f, 0.f), NAME_None,true);
```

# UNITY2UNREAL

## PART 5: COROUTINES vs TIMERS

### Coroutines with IEnumerator

Unity uses **Coroutines** with **IEnumerator** functions and **yield return** statements. They pause execution and resume later without blocking the main thread, perfect for time-based sequences and delays.

```
IEnumerator MyCoroutine()
{
    Debug.Log("Start of Coroutine");
    yield return new
WaitForSeconds(2.0f);
    Debug.Log("End of Coroutine");
}

// Start the coroutine
StartCoroutine(MyCoroutine());
```

### Performance Considerations

⚠️ **GC Pressure:** Using *new WaitForSeconds()* in loops creates garbage collection pressure. **Cache yield instructions** to avoid frequent allocations and performance stutters.

```
// ❌ Bad: Allocates every frame
while (true) {
    yield return new WaitForSeconds(1f);
}

// ✅ Good: Cache the object
var wait = new WaitForSeconds(1f);
while (true) {
    yield return wait;
}
```

**Performance Winner:** Unreal's **FTimerManager** is significantly more performant than Unity's Coroutines due to **zero garbage collection issues** and highly optimized C++ implementation.

### FTimerManager for Simple Delays

Unreal's **FTimerManager** is the direct equivalent to *WaitForSeconds*. It's a highly optimized system for scheduling function calls after delays, accessed through the **UWorld** object.

```
#include "TimerManager.h"

void AMyActor::BeginPlay()
{
    Super::BeginPlay();

    UE_LOG(LogTemp, Warning, TEXT("Starting
Timer"));

    FTimerHandle MyTimerHandle;
    GetWorld()->GetTimerManager().SetTimer(
        MyTimerHandle, this,
&AMyActor::OnTimerEnd,
        2.0f, false);
}

void AMyActor::OnTimerEnd()
{
    UE_LOG(LogTemp, Warning, TEXT("Timer
Complete!"));
}
```

### Async Actions for Complex Tasks

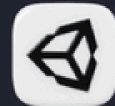
For complex operations, Unreal uses **UBlueprintAsyncActionBase** classes. These create Blueprint nodes with input/output execution pins, similar to complex Unity coroutines but with better performance characteristics.

```
UCLASS()
class UMyAsyncAction : public
UBlueprintAsyncActionBase
{
    GENERATED_BODY()

public:
    UPROPERTY(BlueprintAssignable)
    FMyOutputPin OnSuccess;

    UFUNCTION(BlueprintCallable)
    static UMyAsyncAction* WaitSeveralFrames(
        UObject* WorldContextObject,
        int32 FramesToWait);

    virtual void Activate() override;
};
```

LEARN ALL ABOUT GAME DEV AT OUTSCAL.COM

# UNITY2UNREAL

## PART 6: SERIALIZATION & DATA PERSISTENCE



## UNITY

### [SerializeField] Attribute

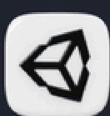
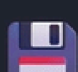
Unity's **[SerializeField]** forces serialization of private fields, making them visible in the Inspector. By default, Unity only serializes public fields and those marked with this attribute.

```
public class PlayerStats : MonoBehaviour
{
    // Private field visible in Inspector
    [SerializeField] private int health = 100;
    [SerializeField] private float speed = 5.0f;

    // This won't appear in Inspector
    private string playerName;

    public void TakeDamage(int damage)
    {
        health -= damage;
    }
}
```

### Save/Load with ScriptableObject

Unity uses **ScriptableObject** for data persistence and JSON serialization for save games. Values persist between scenes and game sessions.

```
[CreateAssetMenu]
public class GameSettings : ScriptableObject
{
    [SerializeField] private int level;
    [SerializeField] private float experience;
}

// Save to JSON
string json = JsonUtility.ToJson(gameData);
File.WriteAllText(savePath, json);
```

**Key Difference:** Unity's approach is **simpler for beginners** but requires explicit attribute marking. **SerializeField** only handles Inspector visibility, not save/load functionality.

## UNREAL ENGINE

### UPROPERTY() Macro

Unreal's **UPROPERTY()** is a powerful reflection system that handles serialization, Blueprint exposure, garbage collection, and editor integration all in one macro.

```
UCLASS()
class MYGAME_API APlayerCharacter : public APawn
{
    GENERATED_BODY()

public:
    // Visible in Blueprint & Details panel
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 Health = 100;

    UPROPERTY(VisibleAnywhere, BlueprintReadOnly)
    float Speed = 5.0f;

private:
    // Not exposed, regular C++ variable
    FString PlayerName;
};
```

### SaveGame Integration

Unreal provides built-in **USaveGame** class with **SaveGame** specifier for automatic persistence. Data automatically saves/loads across game sessions.

```
UCLASS()
class UMyGameSave : public USaveGame
{
    GENERATED_BODY()

public:
    UPROPERTY(SaveGame)
    int32 PlayerLevel;

    UPROPERTY(SaveGame)
    float Experience;
};

// Save automatically handled
UGameplayStatics::SaveGameToSlot(SaveGameInstance,
TEXT("Player1"), 0);
```