

CS354P

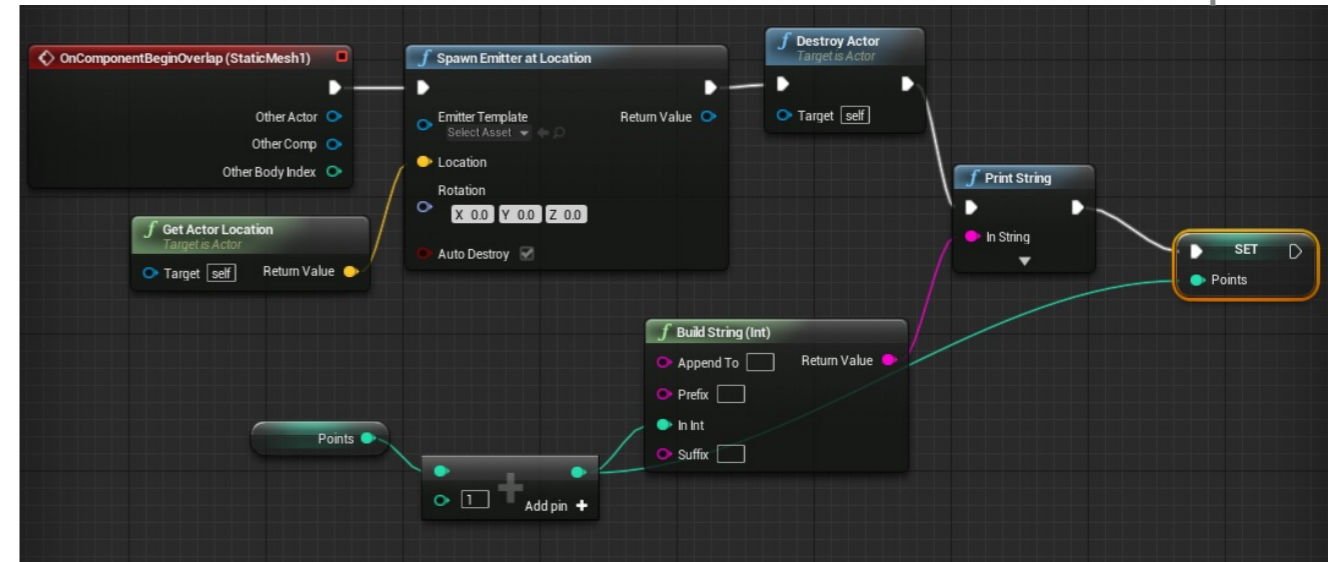
DR SARAH ABRAHAM

C++ AND BLUEPRINTS

GAME SCRIPTING LANGUAGES

- ▶ Most modern game engines assume a C++ base and an in-engine scripting language
- ▶ Performant code written in lower-level language
- ▶ Designer prototyping and less system-critical code written in scripting language

UE4 Blueprint



Godot GDScript

```

01 # This file is part of the Jetpaca game
02 # Copyright (c) 2009-2016 Juan Linietsky, Ariel Manzur
03 # Distributed under the terms of the MIT license (cf. LICENSE.md file)
04
05 extends Area2D
06
07 export(String) var hint="It's dangerous outside. Take this JetPack."
08
09 var side_right = true
10
11 func _process(delta):
12
13     var cc = get_global_pos().x

```

```

// Update is called once per frame
void Update () {

    //this generates a new 'score' string given the states of both variables
    GetComponent<TextMesh>().text = enemyscore.ToString() + " || " + myscore.ToString();
    //this checks if the ball is out of bounds, increments the appropriate score,
    //and resets the ball's position and velocity
    if (ball.transform.position.x > 14){
        myscore++;
        ball.transform.position = new Vector3(7,0,2);
        ball.rigidbody.velocity = new Vector3(0,0,0);
        ball.rigidbody.AddForce(Vector3.right * 200 + Vector3.forward * 100);
    }
    if (ball.transform.position.x < -2){
        enemyscore++;
        ball.transform.position = new Vector3(7,0,2);
        ball.rigidbody.velocity = new Vector3(0,0,0);
        ball.rigidbody.AddForce(Vector3.right * 200 + Vector3.forward * 100);
    }
}
}

```

Unity C#

```

> get_global_pos().x
(1,1)
(-1,1)
acter")

```

C++ AND BLUEPRINTS

- ▶ Blueprints in **native visual scripting language** that is built on top the underlying C++ data structures
- ▶ Blueprint is intended for use by designers and artists
 - ▶ Programmers build out basic functionality in C++ and make it accessible in Blueprints
 - ▶ Designers/artists compose accessible blocks to customize functionality

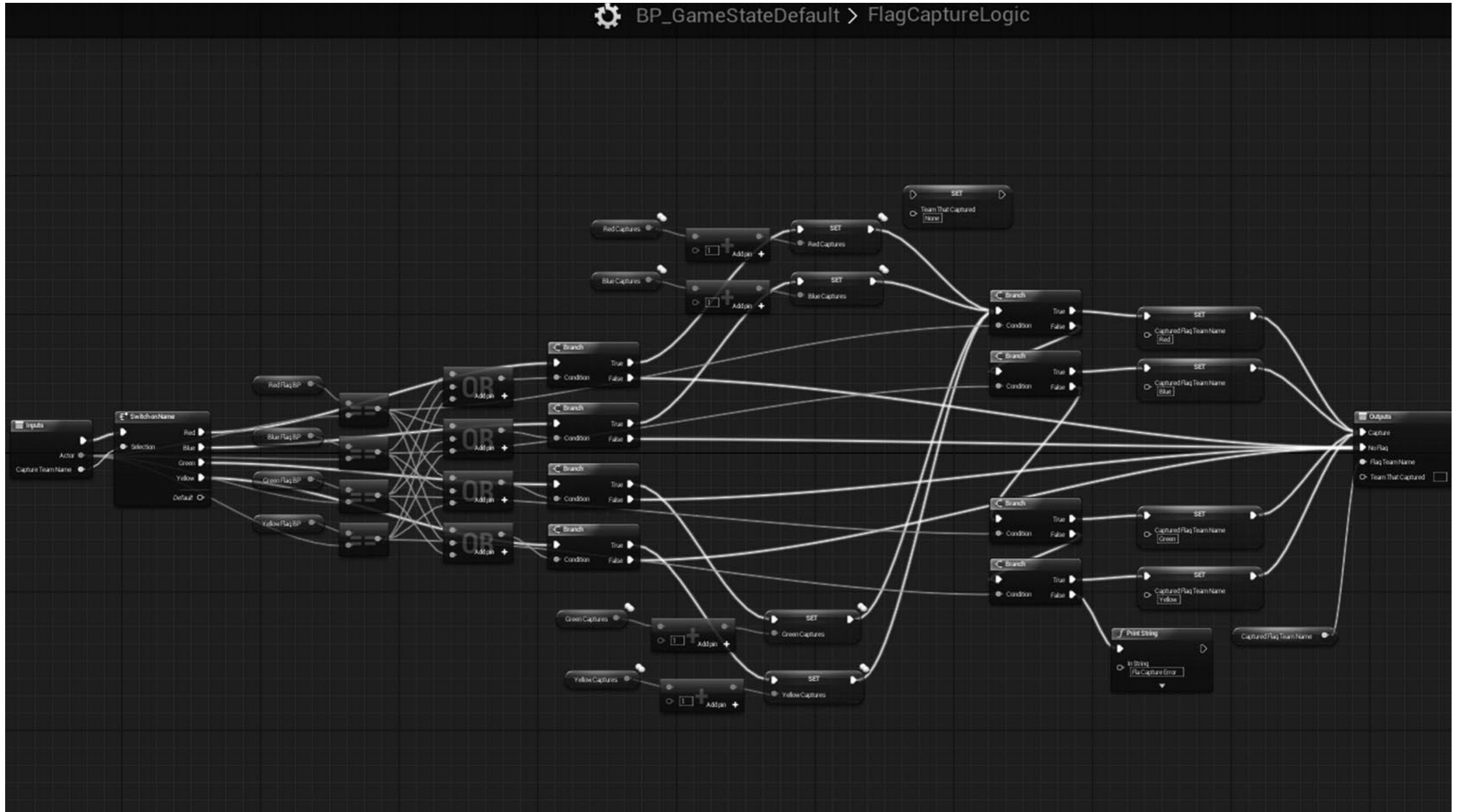
NODE-BASED AND OBJECT-ORIENTED

- ▶ Logical structure of code represented in a visual way
 - ▶ One-way exec pins create order of execution
 - ▶ type pins allow values to be processed and fed into other functionalities
- ▶ Object-oriented node structure matches underlying C++
- ▶ Different nodes provide different functionalities
 - ▶ Incoming and outgoing pin types determined by node

BLUEPRINT LIMITATIONS

- ▶ *Significantly* slower than C++
 - ▶ Can be 25x slower than equivalent C++ code!
- ▶ Reduced functionality
 - ▶ Not all library features are accessible via Blueprint
- ▶ Reduced readability
 - ▶ Visual scripting is faster for prototyping but harder to reason about/maintain

PURE BLUEPRINT EXAMPLE



EXAMPLE UNREAL GAME OBJECT CODE

Enums are BlueprintType making them accessible from BP

```
//Header info here

UENUM(BlueprintType)
enum class ECharacterReactionStateEnum : uint8 {
    HEALTHY          UMETA(DisplayName = "Is Healthy"),
    HIT              UMETA(DisplayName = "Is Hit"),
    DYING            UMETA(DisplayName = "Is Dying"),
    DEAD             UMETA(DisplayName = "Is Dead")
};

UENUM(BlueprintType)
enum class ECharacterStrikeEnum : uint8 {
    LIGHT            UMETA(DisplayName = "Light Hit"),
    HEAVY            UMETA(DisplayName = "Heavy Hit"),
    SPECIAL          UMETA(DisplayName = "Special")
};

DECLARE_DYNAMIC_MULTICAST_DELEGATE(FCharacterActionDelegate);

UCLASS(Blueprintable, config = Game)
class SKAZKA_API ASkazkaCharacter : public ACharacter
{
    GENERATED_BODY()
}
```

Derived class inherits from ACharacter.
Blueprintable makes it accessible as a BP

```
public:
```

```
    ASkazkaCharacter(const FObjectInitializer& ObjectInitializer);
```

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

```
    virtual void Tick(float DeltaSeconds) override;
```

```
    virtual void SetupPlayerInputComponent(UInputComponent* inputComponent)  
    override;
```

```
    virtual void FellOutOfWorld(const class UDamage
```

Standard functionality inherited from
ACharacter (a subclass of APawn)

```
    UFUNCTION(BlueprintImplementableEvent, Category = "Input Events")  
    void move(float value);
```

```
    UFUNCTION(BlueprintImplementableEvent, Category = "Input Events")  
    void jumpStarted();
```

```
    UFUNCTION(BlueprintImplementableEvent, Category = "Input Events")  
    void jumpEnded();
```

```
    UFUNCTION(BlueprintImplementableEvent, Category = "Input Events")  
    void lightAttackStarted();
```

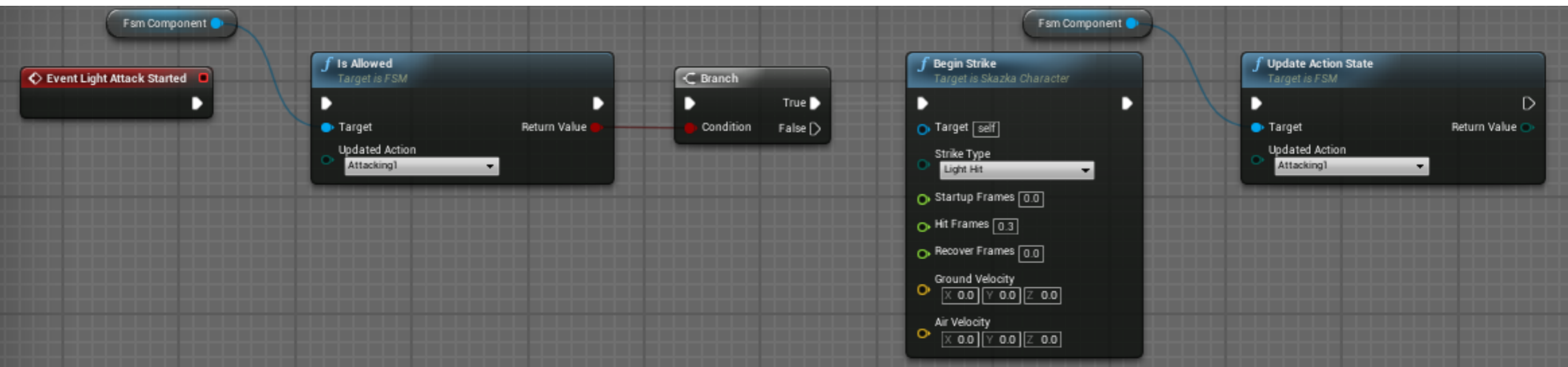
```
    UFUNCTION(BlueprintImplementableEvent, Category = "Input Events")  
    void lightAttackEnded();
```

```
    ...
```

C++ declared events for BP child.
BlueprintImplementableEvents must be
made public.

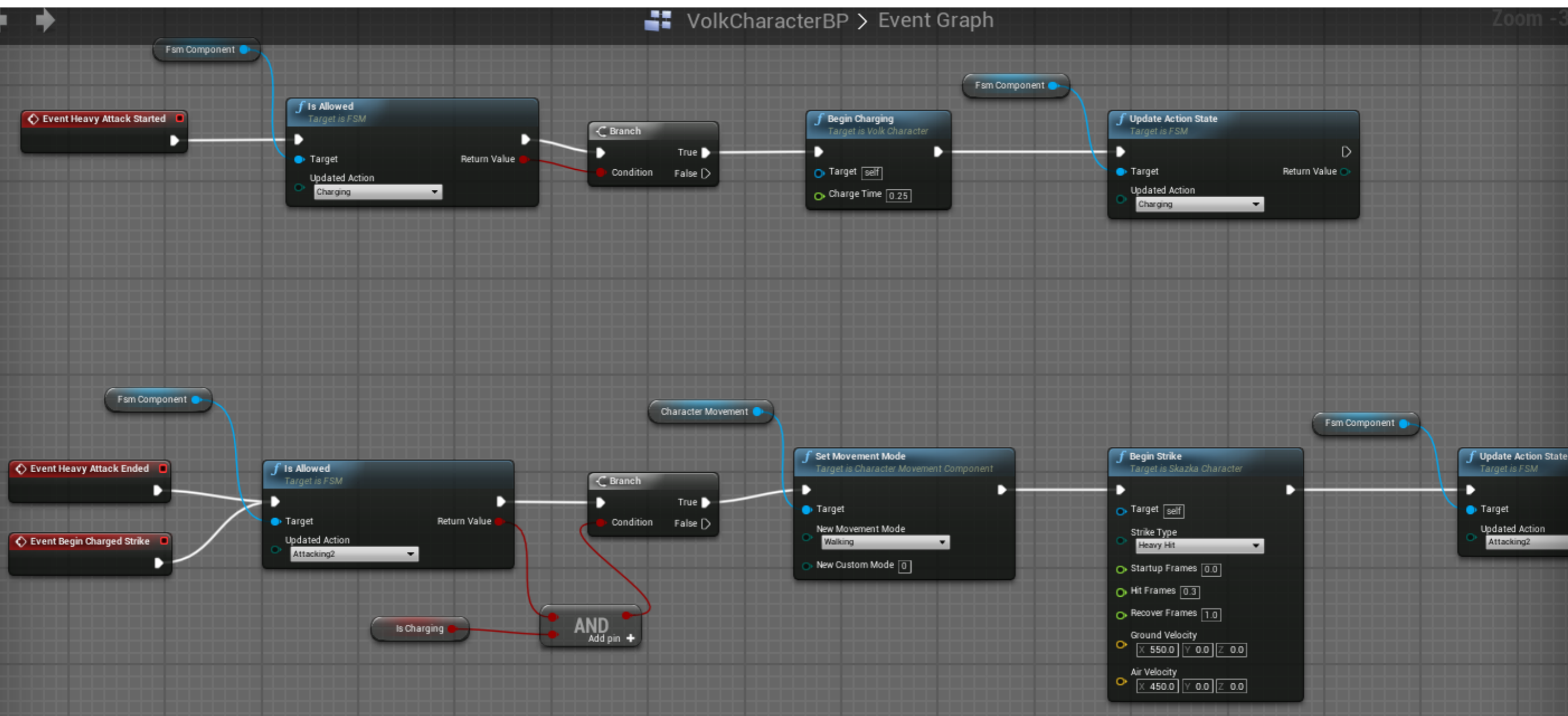
COMBINING C++ AND BLUEPRINT

- ▶ Blueprint classes can extend either another Blueprint class or a C++ class
- ▶ C++ functions and properties can have specifiers that allow them to interact with Blueprint classes



ANOTHER BLUEPRINT EXAMPLE

▶ Character charged attack



SOME FUNCTION SPECIFIERS

- ▶ BlueprintCallable
 - ▶ Function created in C++
 - ▶ Called from either C++ or Blueprint
- ▶ BlueprintImplementableEvent
 - ▶ Function overridden by Blueprint
 - ▶ No body in C++
 - ▶ Autogenerated code includes a `think*` that calls `ProcessEvent`
- ▶ BlueprintNativeEvent
 - ▶ Function has both native C++ and can be overridden by
 - ▶ Blueprint Body is implemented as `[functionname]_Implementation`
 - ▶ Autogenerated code includes **think** to call implementation when necessary

WHAT IS A THUNK?

- ▶ A small subroutine that is called within another subroutine the jumps to another location
 - ▶ Can insert operations into other subroutines
 - ▶ Useful in OOP, where a method can be called by several interfaces
- ▶ Used in UE4 to call into the **Blueprint VM** from the base C++ function
 - ▶ If the Blueprint does not provide this function, does nothing

SOME PROPERTY SPECIFIERS

- ▶ BlueprintReadOnly
 - ▶ Property can be read by Blueprint but not modified
- ▶ BlueprintReadWrite
 - ▶ Property can be read and written from a Blueprint
- ▶ EditAnywhere
 - ▶ Property can be edited by property windows (both archetypes and instances)
- ▶ Native
 - ▶ Property is native to C++
 - ▶ C++ code is responsible for serialization and garbage collection

SOME CHARACTER MOVEMENT PROPERTIES

The image displays the Unreal Engine 4 interface for a character blueprint named 'KatyaCharacterBP'. The left sidebar shows the component hierarchy, including 'CapsuleComponent', 'ArrowComponent', 'Mesh', and 'CharacterMovement'. The central viewport shows the 'Event Graph' with three event nodes: 'Event Light Attack Started', 'Event Heavy Attack Started', and 'Event Heavy Attack Ended'. Each event node is connected to an 'Is Allowed' node, which is linked to the 'Fsm Component'. The 'Event Heavy Attack Ended' node is also connected to a 'SET' node that sets the 'Is Whipping' property. The right sidebar shows the 'Character Movement' settings, divided into sections for 'Jumping / Falling', 'General Settings', 'Walking', and 'Swimming'. Each section contains various numerical and boolean properties.

Character Movement: Jumping / Falling

Jump Z Velocity	570.0
Braking Deceleration Falling	500.0
Air Control	0.5
Air Control Boost Multiplier	2.0
Air Control Boost Velocity Threshold	25.0
Falling Lateral Friction	0.0
Impart Base Velocity X	<input checked="" type="checkbox"/>
Impart Base Velocity Y	<input checked="" type="checkbox"/>
Impart Base Velocity Z	<input checked="" type="checkbox"/>
Impart Base Angular Velocity	<input checked="" type="checkbox"/>
Notify Apex	<input checked="" type="checkbox"/>

Character Movement (General Settings)

Gravity Scale	2.0
Max Acceleration	2048.0
Braking Friction Factor	2.0
Braking Friction	0.0
Use Separate Braking Friction	<input type="checkbox"/>
Crouched Half Height	40.0
Mass	130.0
Default Land Movement Mode	Walking
Default Water Movement Mode	Swimming

Character Movement: Walking

Max Step Height	45.0
Walkable Floor Angle	45.0
Walkable Floor Z	0.707107
Ground Friction	35.0
Max Walk Speed	600.0
Max Walk Speed Crouched	300.0
Min Analog Walk Speed	0.0
Braking Deceleration Walking	2048.0
Sweep While Nav Walking	<input checked="" type="checkbox"/>
Can Walk Off Ledges	<input checked="" type="checkbox"/>
Can Walk Off Ledges when Crouching	<input checked="" type="checkbox"/>
Maintain Horizontal Ground Velocity	<input checked="" type="checkbox"/>
Ignore Base Rotation	<input type="checkbox"/>

Character Movement: Swimming

Max Swim Speed	300.0
Braking Deceleration Swimming	0.0
Buoyancy	1.0

COMPILING C++ AND BLUEPRINT

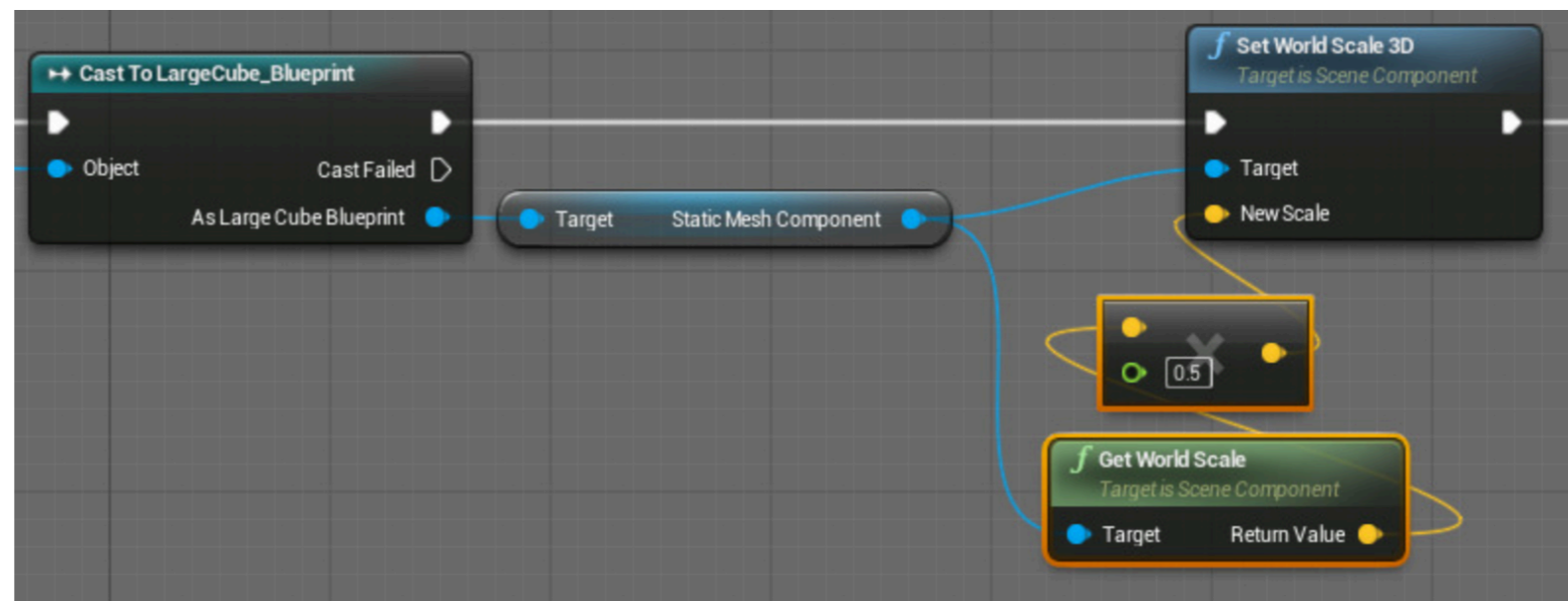
- ▶ C++ can be Hot Reloaded
 - ▶ Allows compiling of C++ from both IDE or Editor without shutting down the Editor
 - ▶ Note: Must build and run in IDE to use C++ breakpoints during debugging
- ▶ Blueprints must also be compiled
 - ▶ Save and compile BPs before running

CASTING WITHIN BLUEPRINT

- ▶ Possible to cast objects to other types
- ▶ C++ way:

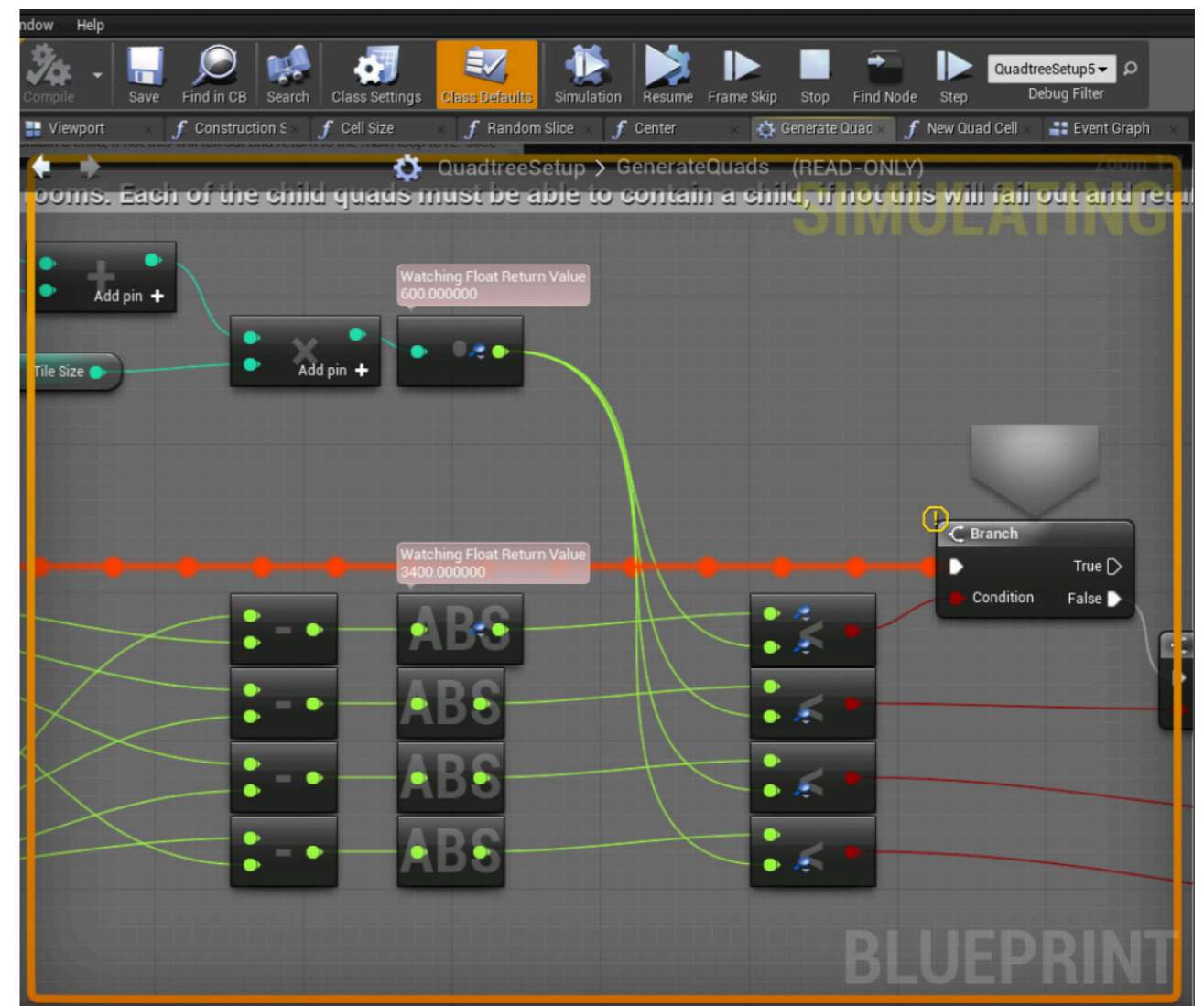
```
AMyActor* myActor = Cast<AMyActor>(actor);  
  
if (myActor) { ... }
```

- ▶ Blueprint way:



BLUEPRINT DEBUGGING

- ▶ Can debug Blueprints in similar ways to C++
 - ▶ Breakpoints
 - ▶ Call stack
 - ▶ Execution Trace
 - ▶ Print statements
 - ▶ Visual Debugger



Example of visual debugger showing game's current execution

WHEN TO USE C++ VERSUS BLUEPRINT?

- ▶ Only hard rule is that Blueprint won't be as performant and is less expressive
 - ▶ Lots of flexibility where the dividing line should be depending on team
- ▶ In general, I may do some initial prototyping in Blueprint and compose the high level functionality in Blueprint, but I prefer to do most of the work in code
 - ▶ Cleaner and more maintainable even when performance isn't a big issue
 - ▶ Easier to reconstruct if UE4 decides to eat your BP