CS354P

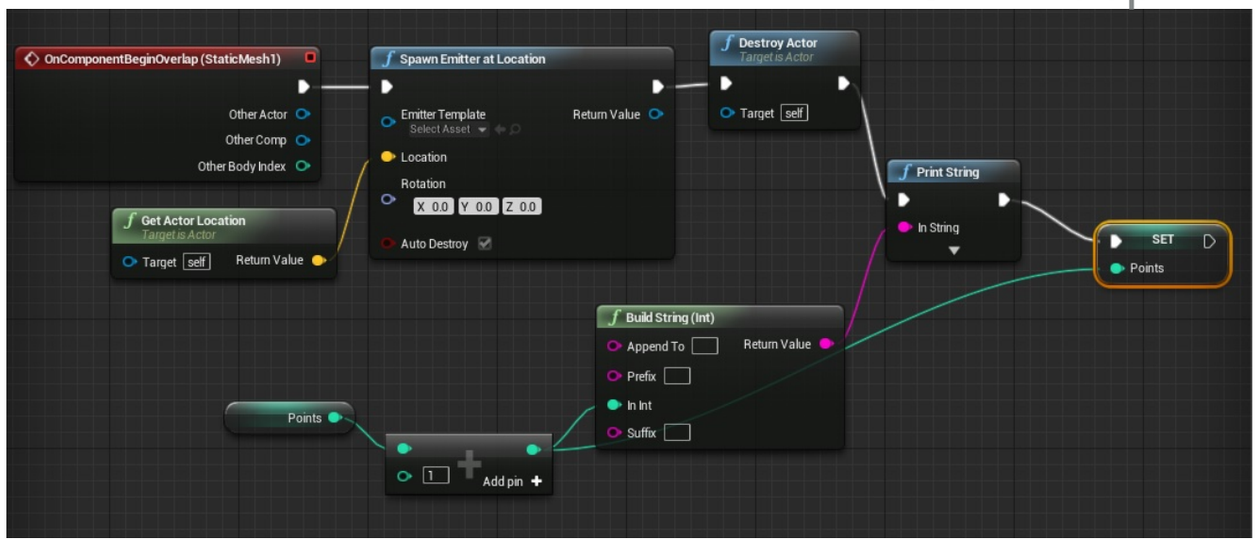
DR SARAH ABRAHAM

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

```
# This file is part of the Jetpaca game
# Copyright (c) 2009-2016 Juan Linietsky, Ariel Manzur
# Distributed under the terms of the MIT license (cf. LICENSE.md file)

extends Area2D

export(String) var hint="It's dangerous outside. Take this JetPack."

var side_right = true

func _process(delta):
```

Unity C#

```
// 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);
    }

}
```

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

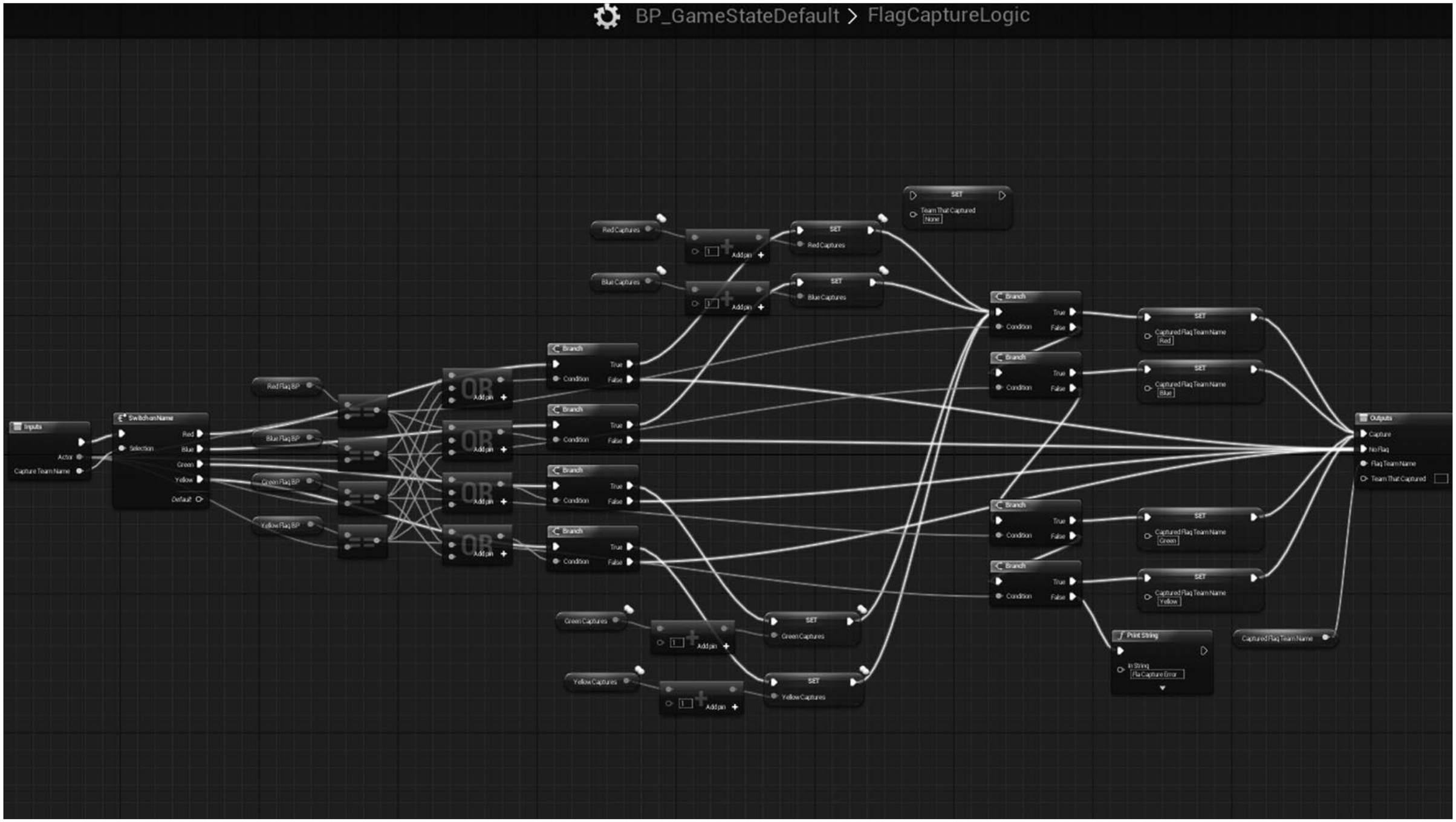
NODE-BASED AND OBJECT-ORIENTED

- 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 UDam
```

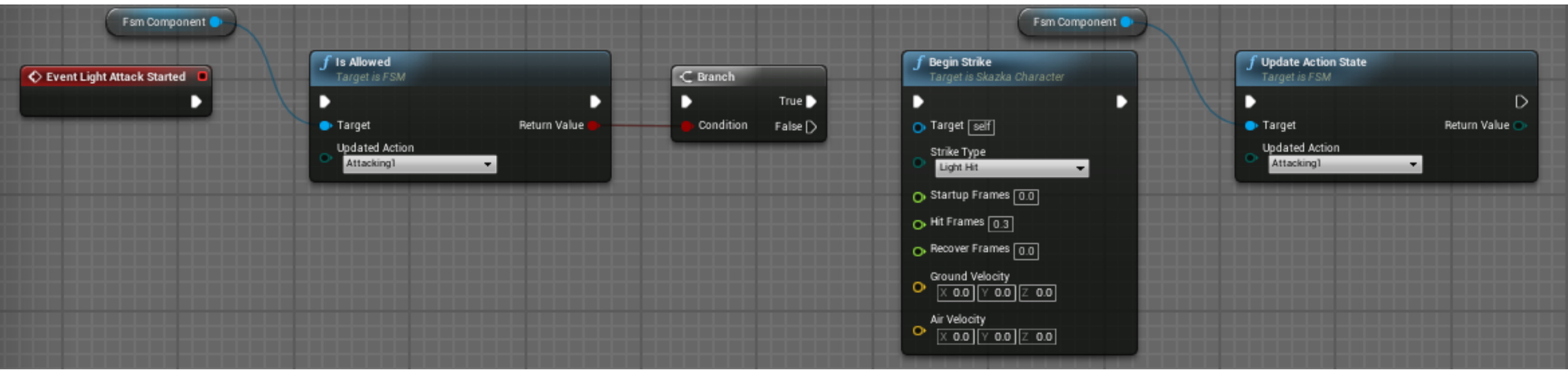
Standard functionality inherited from ACharacter (a subclass of APawn)

```
    UFUNCTION(BlueprintImplementableEvent, Catego
        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.

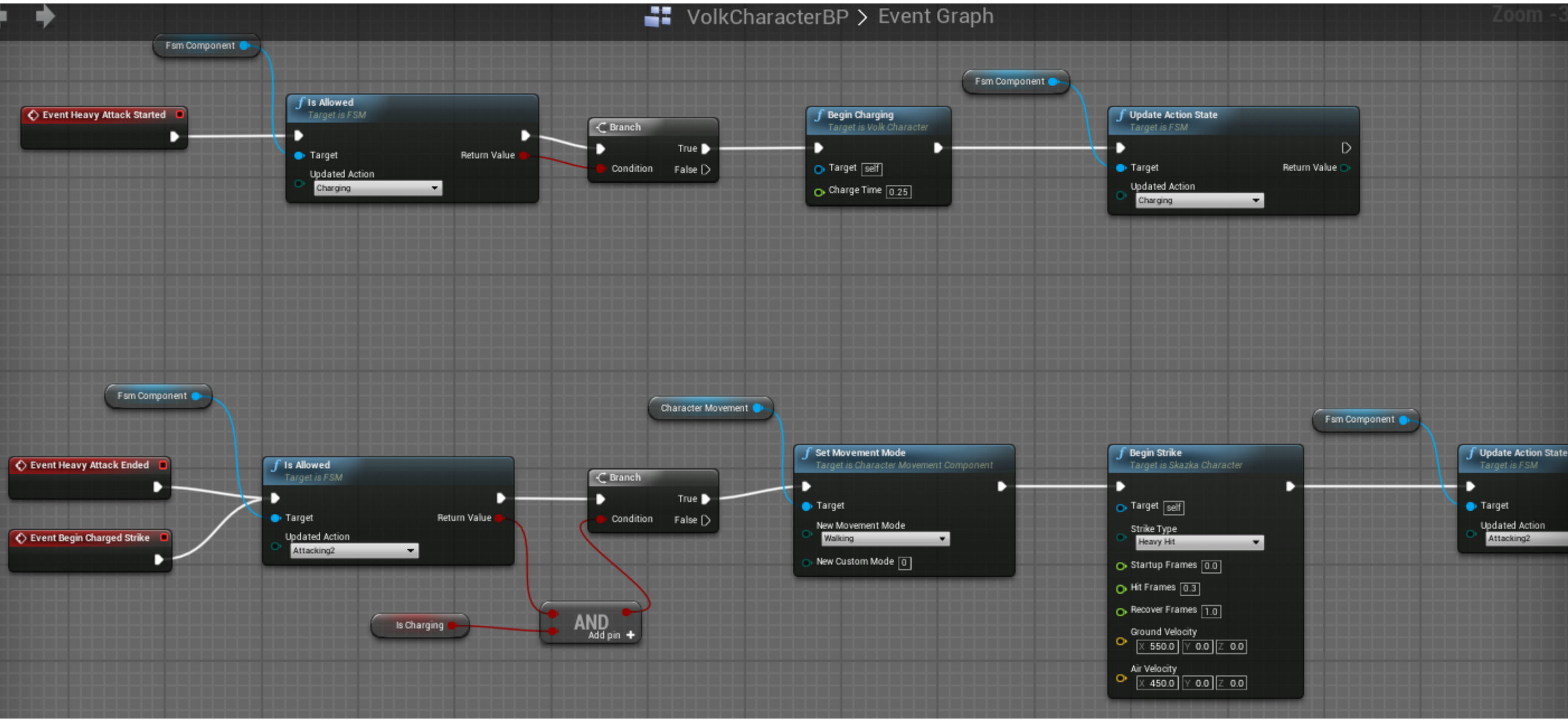
COMBINING C++ AND BLUEPRINT

- 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 thunk* that calls ProcessEvent
- BlueprintNativeEvent
  - Function has both native C++ and can be overridden by
  - Blueprint Body is implemented as [functionname]_Implementation
  - Autogenerated code includes **thunk** to call implementation when necessary

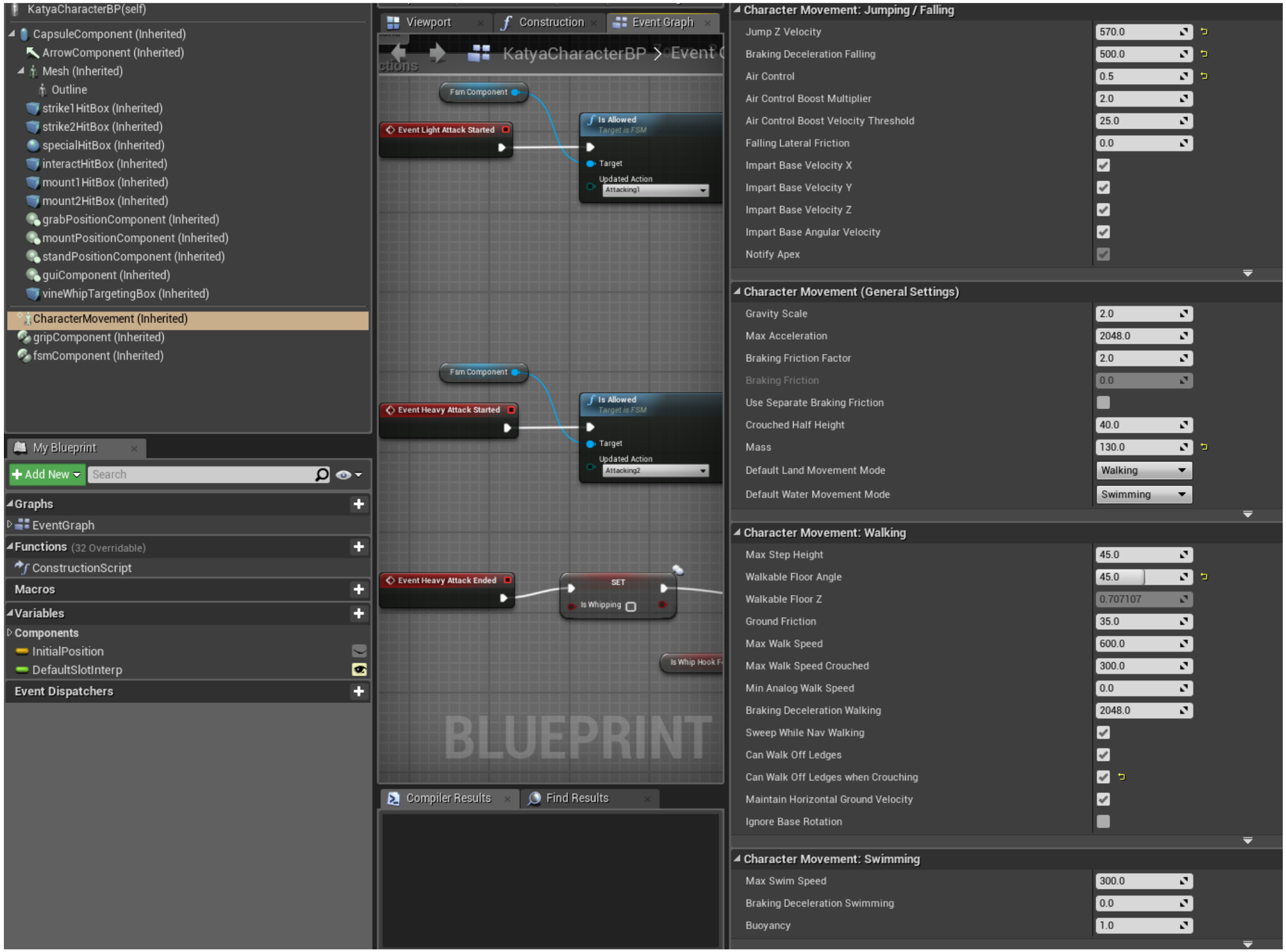
WHAT IS A THUNK?

- 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 Unreal 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



COMPILING C++ AND BLUEPRINT

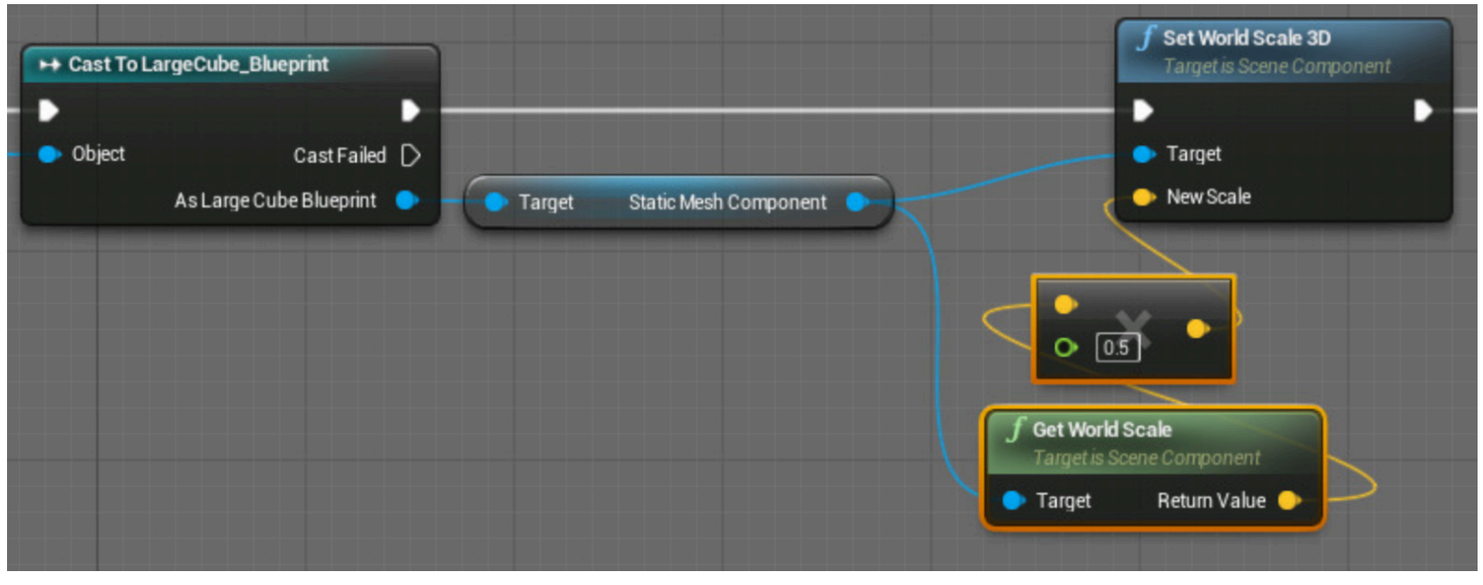
- C++ can be Hot Reloaded
  - Allows compiling of C++ from both IDE or Editor without shutting down the Editor
    - Note: Must build and run in IDE to use C++ breakpoints during debugging
- Blueprints must also be compiled
  - Save and compile BPs before running

CASTING WITHIN BLUEPRINT

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

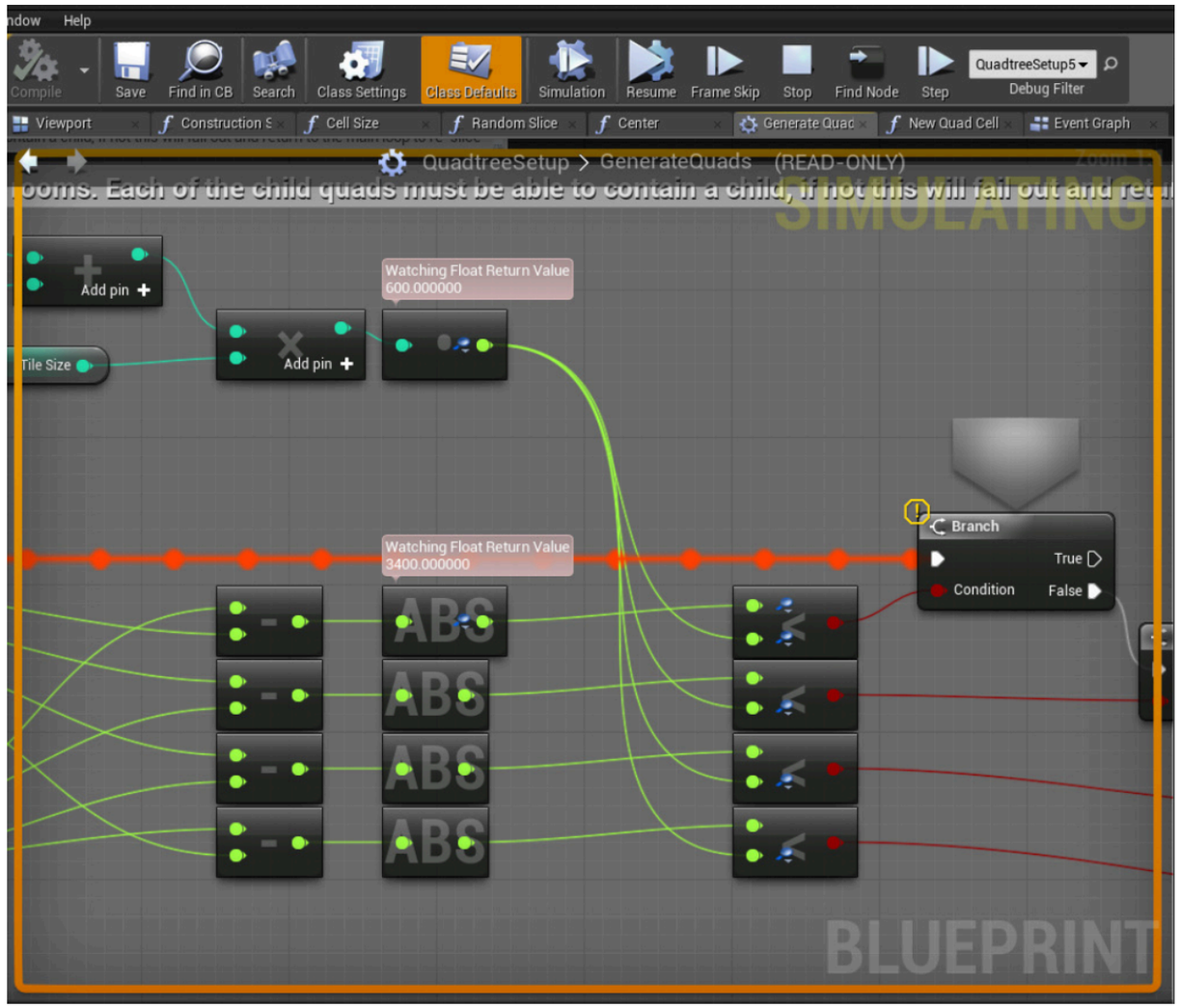
```
AMyActor* myActor = Cast<AMyActor>(actor);

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 Unreal decides to eat your BP