Blue Gravity Interview Task

Skateboarding simulator game

Andrés Adarve

Movement System - 10h

The character movement system, in order to simulate as closely as possible the behavior of a real skateboard, primarily relies on configuring parameters of the Character Movement Component to generate a low-friction effect that keeps the character in motion while gradually losing momentum.

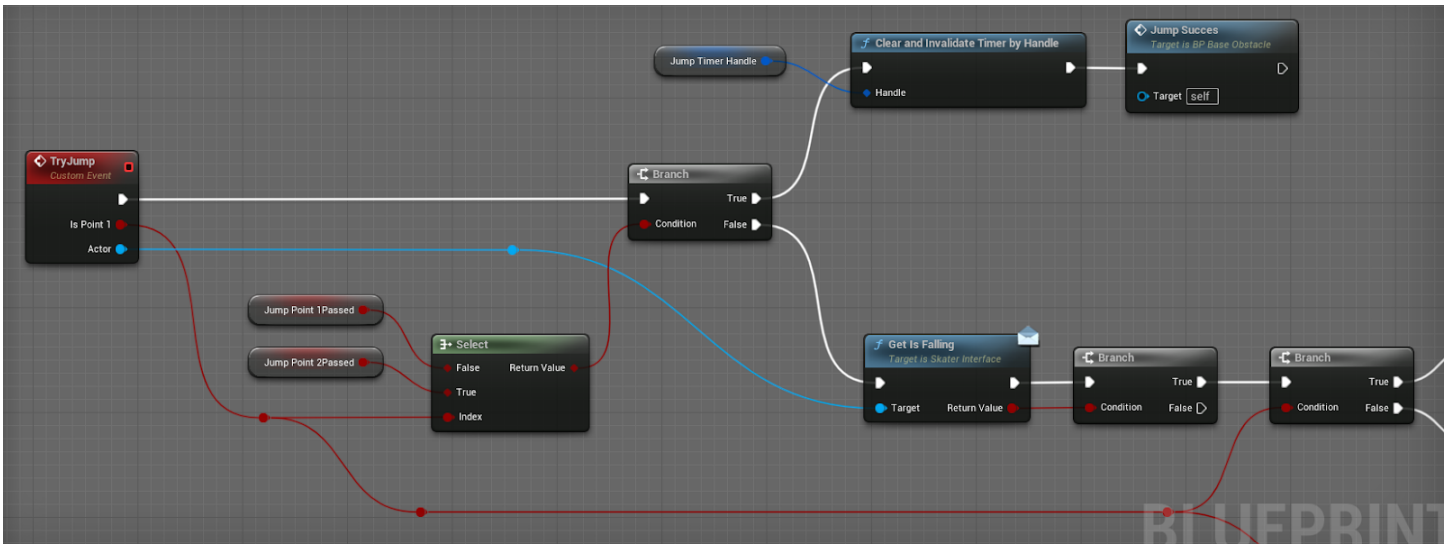
The base class SkateboardingCharacter.cpp (derived from the default third-person class) defines the movement. Pressing the W key propels the character in the direction of the skateboard's forward vector, while the S key gradually reduces its speed. The A and D keys rotate the character to a greater or lesser extent depending on the movement state (rotation is reduced while the character is propelled, simulating the increased difficulty of turning at higher speeds).

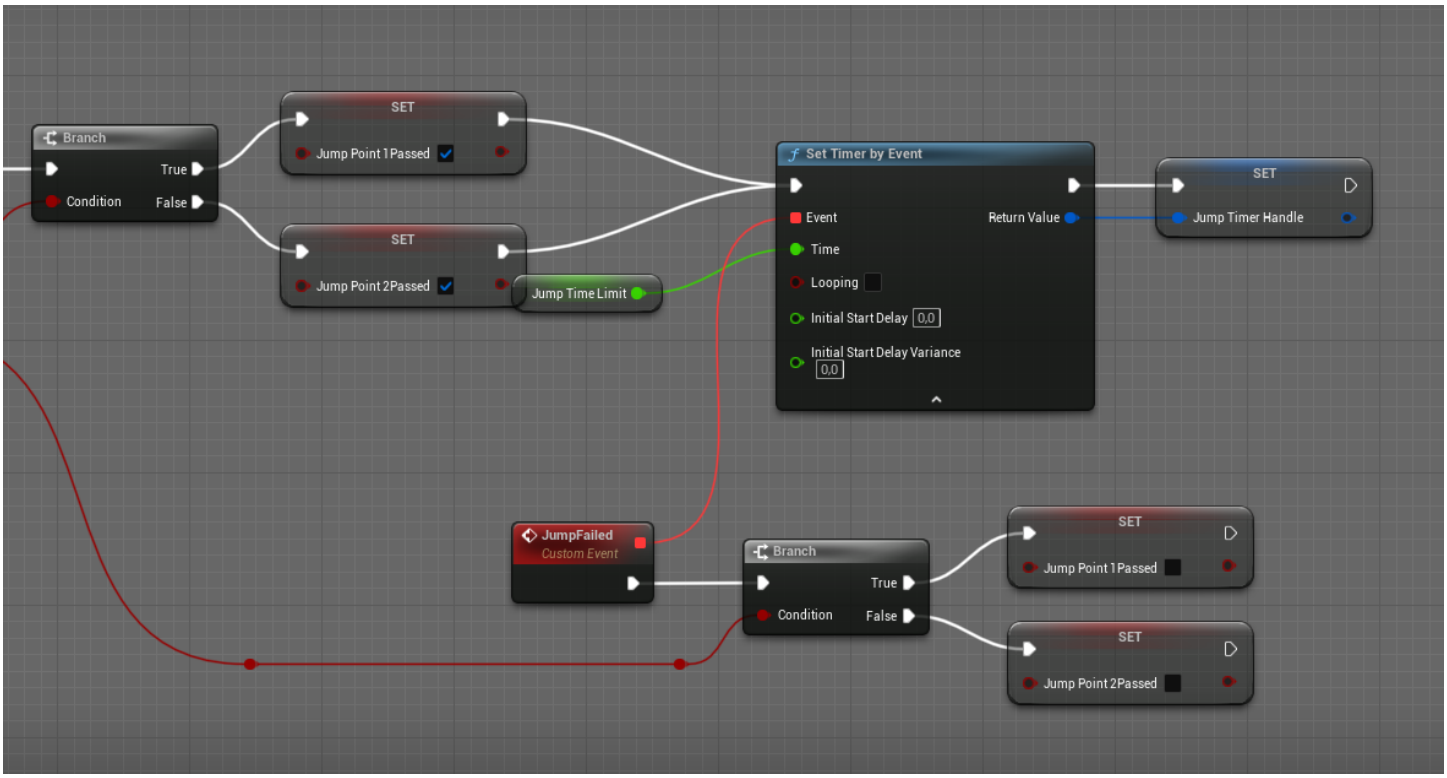
```
void ASkateBoardingCharacter::Push()
{
    if (Controller != nullptr)
    {
        // get forward vector
        const FVector ForwardDirection = Skateboard->GetForwardVector();

        if(!GetMovementComponent()->IsFalling())
        {
            MoveForwardValue = FMath::Lerp( A: MoveForwardValue, B: 1.0f, MoveForwardAlpha);
            AddMovementInput(ForwardDirection, MoveForwardValue);
        }
    }
}
```

```cpp
void ASkateBoardingCharacter::TurnDirection(const FInputActionValue& Value)
{
    // input is a Vector
    float MovementValue = Value.Get<float>();

    if (Controller != nullptr)
    {
        /* Rotate the pawn based on their state, different rotation values when pushing, falling and not doin
         */
        if(GetMovementComponent()->IsFalling())
        {
            MoveRightValue = MovementValue*MoveRightAirMultiplier;
            TurnLeftRight();
        }else
        {
            if(MoveForwardValue>0.0f) //Pushing forward
            {
                MoveRightValue = MovementValue*MoveRightMultiplier;
                TurnLeftRight();
            }else
            {
                MoveRightValue = MovementValue*MoveRightNoImpulseMultiplier;
                const FRotator direction = TurnLeftRight();
                GetMovementComponent()->Velocity = direction.RotateVector(GetMovementComponent()->Velocity);
            }
        }
    }
}
```

Points and Tricks System - 5h

To create a scalable points system for potential future updates, I created an Enum named E_Trick. This Enum comprises a string to assign a name to the trick and an integer being the score obtained upon performing the trick. Obstacles designed for jumping feature a mesh along with two box collisions. When the character crosses (OnEndOverlap) one of these collisions, a timer is activated. If the player crosses the other collision before the timer expires, the trick is considered executed. Subsequently, an E_Trick data type is transmitted to the player state to increment the score in the record and display information about the executed trick on the interface.


Jump Timer Handle
Clear and Invalidate Timer by Handle
Handle
Jump Succes
Target is BP Base Obstacle
Target self
TryJump
Custom Event
Is Point 1
Actor
Branch
True
Condition
False
Jump Point 1Passed
Jump Point 2Passed
Select
False
Return Value
True
Index
Get Is Falling
Target is Skater Interface
Target
Return Value
Branch
True
Condition
False
Branch
True
Condition
False


Branch
True
Condition
False
SET
Jump Point 1Passed
SET
Jump Point 2Passed
Jump Time Limit
Set Timer by Event
Event
Return Value
Time
Looping
Initial Start Delay
0,0
Initial Start Delay Variance
0,0
SET
Jump Timer Handle
JumpFailed
Custom Event
Branch
True
Condition
False
SET
Jump Point 1Passed
SET
Jump Point 2Passed

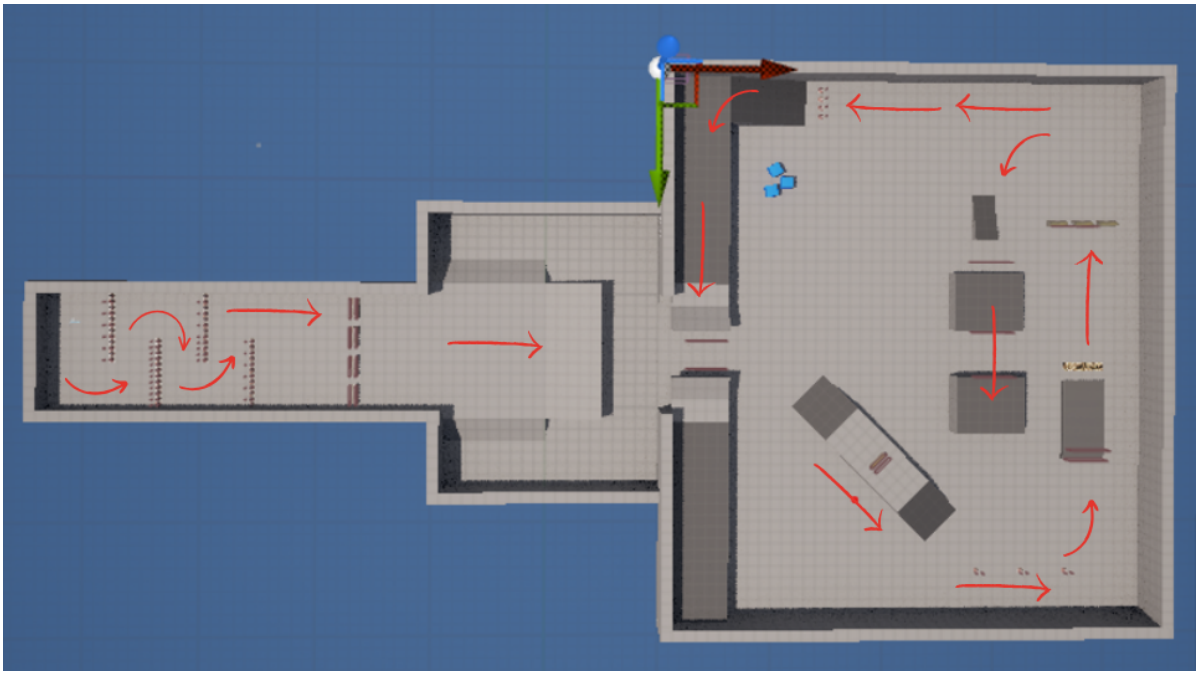
Interface - 1h

I designed the interface with fixed elements positioned in the corners to maximize the central screen space, allowing the player to focus on the environment and obstacles. Meanwhile, the trick names and score are displayed in the lower center to ensure they stand out when shown.

Level Design 4h

The level design is shaped to naturally guide the player using the line of sight. Obstacles and ramps are strategically placed to orient themselves in a manner that forms lines leading the player in the desired direction, employing the Directional Line Pattern.

Obstacles are also strategically placed to teach the player the mechanics in a natural way without the need to use long texts.



Final Remarks

Everything in this project was created during the interview period.

During the process, I decided to develop the basic character movement functions in C++, aiming to optimize performance, as these functions were expected to have the highest number of calls during execution. Once the basic system was implemented, it took me much longer to fine-tune the variables of the Character Movement Component to achieve satisfactory movement that closely resembled real skateboarding.

Regarding obstacles, the simplest method I conceived to detect a simple jump over them was by using two collisions, providing a time frame to pass through the second one after crossing the first. Although it works properly, I acknowledge that it might occasionally lead to errors

depending on the jump angle, collision placements, and player movement. Hence, further testing would be necessary to identify and address these potential issues.

I designed the entire code to be easily scalable and modifiable, utilizing interfaces instead of castings to obtain information from certain classes. This ensures that replacing or modifying these classes in the future would not impact other classes.

Despite achieving a satisfactory result within the development timeframe, I would have liked to include more features, such as simulating falls when the skateboard's orientation upon landing from a jump does not match the character's velocity direction, or automatically accelerating the character based on the slope of the ground. These features will be considered for future additions to expand the prototype.

Additionally, the movement system occasionally presents peculiar behavior due to physics, where collisions with other objects or specific key combinations can cause the character to slide with orientation mismatched to the movement direction. Linking these orientation changes with a falling function could render the movement system more realistic.