

# CryptRaider

Проект на Мишел Хамид F101299

Същност и съдържание на проекта

Играта CryptRaider представлява малка treasure hunt игра. Използван е пакет от асети с тематиката на тъмница и/или гробница. Съдържа главно механики за отваряне на врати след решаване на пъзелите. Реализирани са по различни начини. Проектът съдъра както механики, реализирани със C++ код, така и механики, реализирани с Blueprint.

## C++ Класове

**Actor Component** - Базов клас за компоненти, който дефинира поведение, което може да се преизползва и добавя към други класове.

**Scene Component** - подобно на actor component, поддържа "закачане" (може да се закача за други scene component). Има трансформации, но не поддържа рендериране и няма функции за колизии.

**UPROPERTY** - прави променливите видими в editor средата

- **EditAnywhere** - видима и може да бъде променена от средата .

**UFUNCTION** - прави методите видими в editor средата

- **BlueprintCallable** - методите могат да се извикват в Blueprint.

**FVector** - Вектор в триизмерното пространство, съставен от компоненти (x, y, z) с точност на плаваща запетая.

**FRotator** - Контейнер за информация за ротация. Всички стойности на въртене се съхраняват в градуси.

**UPhysicalHandleComponent** - компонент, който служи за местене на физически обекти;

**AActor** - базовият клас на Object, който може да бъде поставен и създаден в ниво.

**FHitResult** - структура, която съдържа информация за едно попадение на trace/sweep, като точка на удар и повърхностна норма в тази точка.

**FCollisionShape** - форми на колизии, поддържа формите сфера, капсула, кутия или линия

**FName** - стринг, case-insensitive, не могат да се манипулират и не могат да бъдат изменени.

Actor Components в играта:

- Mover - служи за отместване на обекти;
- Rotator Actor - служи за завъртане на обекти.

Scene Components в играта:

- Grabber - служи за "хващане" на обекти.

Collision Components в играта (разписани на C++):

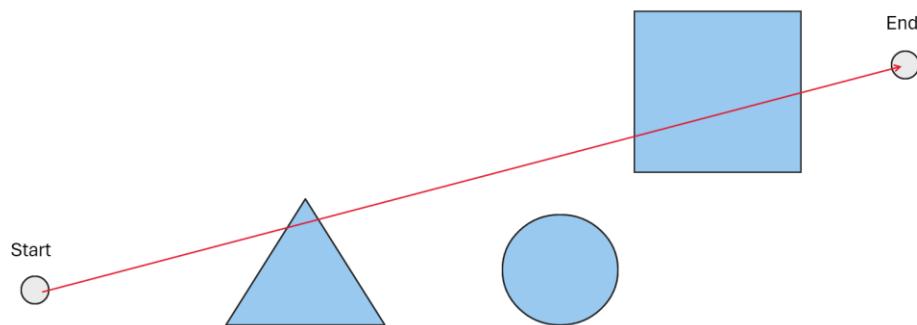
- TriggerComponent - използва се заедно с Mover, служи за изместване на обектите;
- RotatorComponent - използва се заедно с RotatorActor, служи за ротиране на обектите.

## Line Tracing and Geomerty Trace(shape trace)

### Line Tracing

Line tracing е невидима линия или лъч, която се изчертава от начална позиция до крайна позиция в игров свят. Използва се чрез "хвърляне" на линия или лъч и се проверява дали пресича обекти или равнини по пътя. Често се използва за проверяване на колизии, събиране на информация за обекта, при който е настъпила колизия и задействане на различни действия и събития в играта.

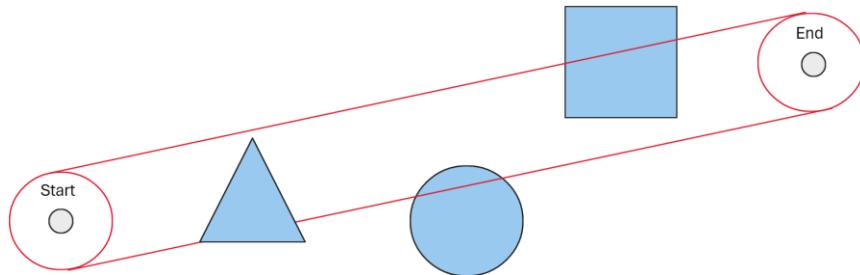
Line trace съдържа стартова позиция и крайна позиция, които определят траекторията на изчертаването. Може да се зададе да пътува във всякаква посока и разстояние, в зависимост от нуждите на играта. Когато се засече пресичане с обект или равнина може да се изведе информация за "удареният/ните" обект/и като локация, ротация и други свойства.



### Geomerty Trace(shape trace)

Line Tracing-а е прекалено прецизен в дадени моменти, за това има алтернатива, която работи перфектно за този проект: Geomtry Tracing.

Същността е същата, но визуализацията е малко по-различна:



Така може да се засичат повече обекти или повърхности, защото радиусът на засичане е по-голям.

Обектите в играта имат нещо наречено Trace Channels. Тези свойства определят как обектите реагират при появя на колизия.

В проекта е направен персонализиран trace channel, който е свързан с класа, отговорен за грабване на обекти - Grabber.

Engine - Collision

Set up and modify collision settings.

These settings are saved in DefaultEngine.ini, which is currently writable.

Object Channels

You can have up to 18 custom channels including object and trace channels. This is the list of object types for your project. If you delete an object type that is being used by the game, any uses of that type will revert to WorldStatic.

Name	Default Response
Projectile	Block

Trace Channels

You can have up to 18 custom channels including object and trace channels. This is the list of trace channels for your project. If you delete a trace channel that is being used by the game, the behavior of the trace is undefined.

Name	Default Response
Grabber	Ignore

Preset

Export... Import... New Object Channel... Edit... Delete... New Trace Channel... Edit... Delete...

Схема за припокриване при колизия с Geometry Tracing:

		OBJECT A		
		Ignore	Overlap	Block
OBJECT B	Ignore	IGNORE	IGNORE	IGNORE
	Overlap	IGNORE	OVERLAP	OVERLAP
	Block	IGNORE	OVERLAP	BLOCK

C++

Actor Components

Mover



Mover.h

Променливи:

- FVector MoveOffset – UPROPERTY, определя с колко да се измести обекта. Задава се от editor средата. Вектор с три стойности;
- float MoveTime - времето за изместване, в секунди;
- bool ShouldMove - помощна булева променлива, използва се в методът SetShouldMove;
- FVector OriginalLocation - за запазване на първоначалните координати на обекта. Вектор с три стойности.

Методи:

- void SetShouldMove(bool ShouldMove) - помощен метод, служи за свързване на два класа. Променя стойността на променливата ShouldMove.

Mover.cpp

```
// Called when the game starts
void UMover::BeginPlay()
{
    Super::BeginPlay();

    OriginalLocation = GetOwner() -> GetActorLocation();
}
```

При започване на играта веднага взимаме първоначалните координати на обекта, който искаме да преместим. GetOwner намира актьора, към когото е прикачен компонентът Mover , и взима pointer към него. Чрез -> оператора взимаме директно стойността, в случая координатите, на обекта и ги записваме в променливата OriginalLocation.

```
// Called every frame
void UMover::TickComponent(float DeltaTime, ELevelTick TickType, FActorComponentTickFunction* ThisTickFunction)
{
    Super::TickComponent(DeltaTime, TickType, ThisTickFunction);

    if(ShouldMove){
        FVector CurrentLocation = GetOwner() -> GetActorLocation();
        FVector TargetLocation = OriginalLocation + MoveOffset;
        float Speed = FVector::Distance(OriginalLocation, TargetLocation) / MoveTime;

        FVector NewLocation = FMath::VInterpConstantTo(CurrentLocation, TargetLocation, DeltaTime, Speed);
        GetOwner() -> SetActorLocation(NewLocation);
    }
}
```

Методът TickComponent се извиква всеки frame към текущия актьор. Кодът в метода се изпълнява на всеки frame. Преди да се изпълни преместването, се проверява дали компонентът трябва да се премести.

- true - извършва се преместването
  - Създава се помощна променлива CurrentLocation, която държи текущите координати на обекта.;
  - Създава се втора, помощна променлива, която ще държи координатите, на които трябва да се премести обектът. Координатите, на които трябва да се премести обектът са сбор от координатите на текущата локация и и координатите на първоначалната локация;
  - Създава се трета, помощна променлива, която ще държи скоростта. Използва се формулата за скорост:  $V = S / T$ .
    - Distance(FVector& V1, FVector& V2) е метод, който принадлежи на класа FVector. Служи за измерване на разстоянието между два вектора.
  - Последната помощна променлива е NewLocation. VInterpConstantTo(Current, Target, DeltaTime, Speed) е метод на класът FMath, който служи за интерполиране от текущите координати до желаните координати непрекъснато'
  - Задаване на новите координати с помощта на метода SetActorLocation(FVector& NewLocation).

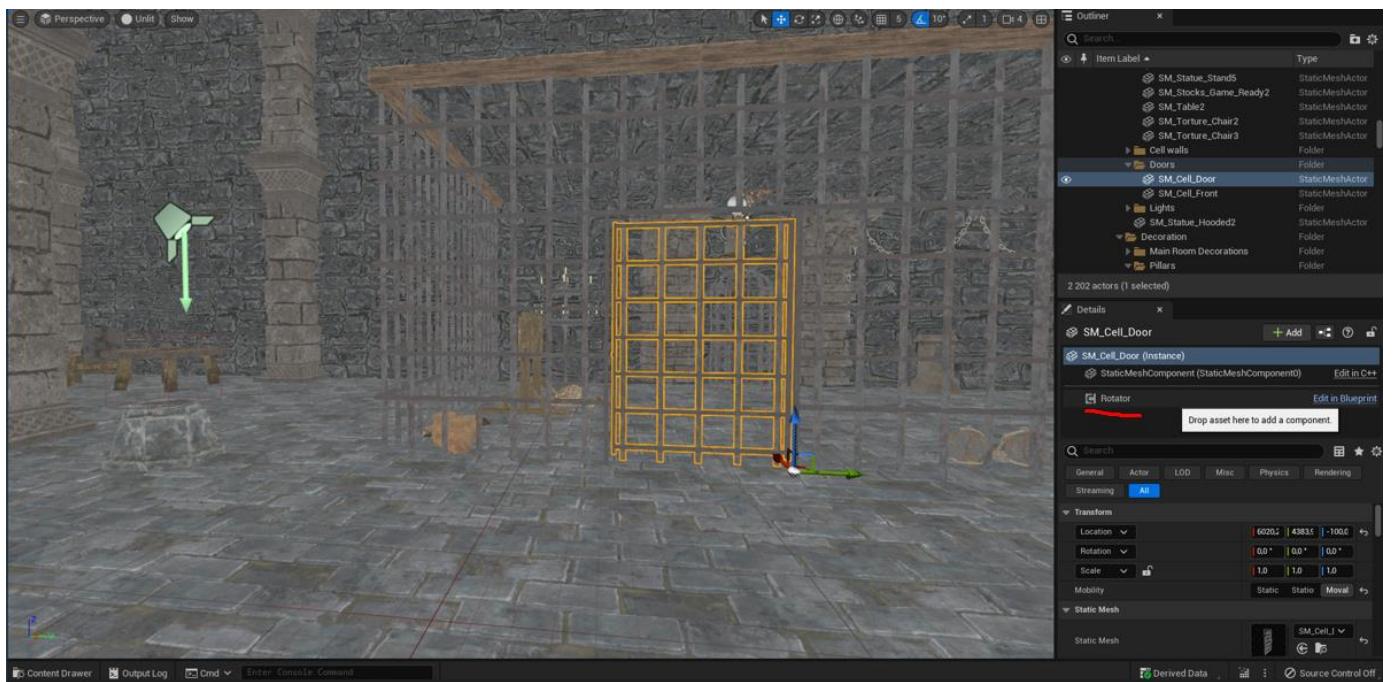
- false - нищо не се случва.

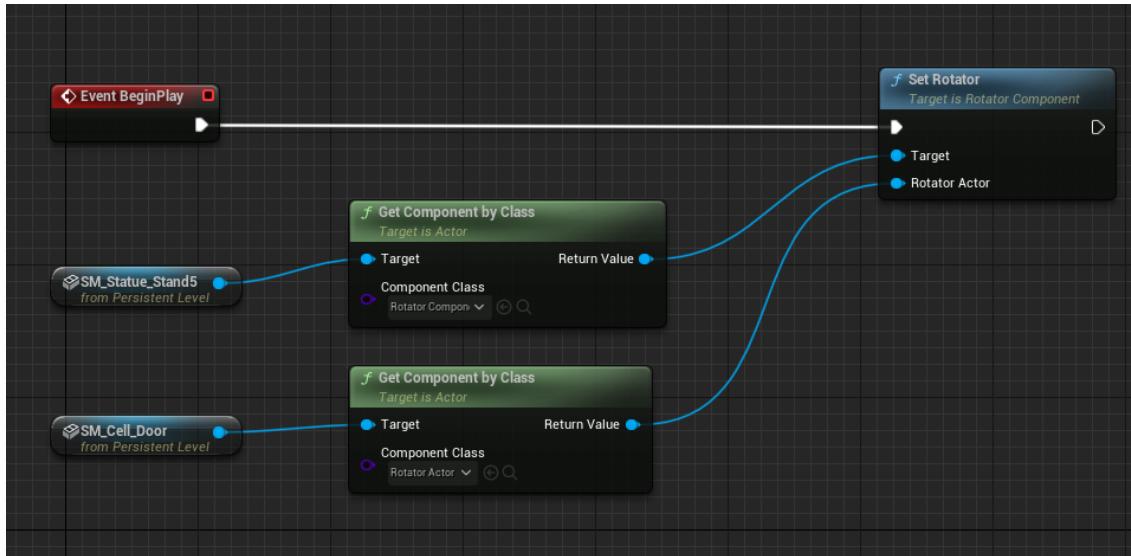
```
void UMover::SetShouldMove(bool NewShouldMove)
{
    ShouldMove = NewShouldMove;
}
```

- метод, който служи за задаване на

променливата ShouldMove

## Rotator Actor





RotatorActor.h

Променливи:

- FRotator RotationOffset – UPROPERTY, определя с колко градуса да се завърти обекта
  - #include "Math/Rotator.h";
- Frotator OriginalRotation - първоначалната ротация на обекта при започване на играта;
- float RotationTime - времето за ротация на обекта в секунди;
- bool ShouldRotate - помощна променлива, използва се в метода SetShouldRotate.

Методи:

- void SetShouldRotate(ShouldRotate).

RotatorActor.cpp

```

// Called when the game starts
void URotatorActor::BeginPlay()
{
    Super::BeginPlay();
    OriginalRotation = GetOwner() -> GetActorRotation();
}

```

При започване на играта се взима първоначалната ротация на обекта и се записва в променливата OriginalRotation. GetOwner намира актьора, към когото е прикачен компонентът RotatorActor, и взима pointer към него. Чрез -> оператора се взима директно стойността, в случая стойността на ротацията, на обекта и се записва в променливата OriginalRotation.

```
// Called every frame
void URotatorActor::TickComponent(float DeltaTime, ELevelTick TickType, FActorComponentTickFunction* ThisTickFunction)
{
    Super::TickComponent(DeltaTime, TickType, ThisTickFunction);

    if(ShouldRotate)
    {
        UE_LOG(LogTemp, Display, TEXT("Rotating"));

        FRotator CurrentRotation = GetOwner() -> GetActorRotation();
        FRotator TargetRotation = OriginalRotation + RotationOffset;
        float Speed = RotationOffset.Euler().Length() / RotationTime;

        FRotator NewRotation = FMath::RInterpConstantTo(CurrentRotation, TargetRotation, DeltaTime, Speed);
        GetOwner() -> SetActorRelativeRotation(NewRotation);
    }
}
```

- След започване на играта, методът TickComponent се извиква всеки фрейм. Преди да се извърши ротацията се проверява дали обектът трябва да се завърти:

- true :
  - UE\_LOG() - принтира съобщение в конзолата. Използва се за тестване, защото понякога UE5 не изпълнява нормално командите;
  - Създава се първата помощна променлива CurrentRotation от тип FRotator. В нея се записва текущата ротация на обекта, подобно на OriginalRotation;
  - Създава се втора помощна променлива TargetRotation от тип FRotator. В нея се изчислява желаната ротация - това става като към текущата стойност на ротацията на обекта се добави стойността на RotationOffset. С други думи се изчисляват градусите, на които ще се завърти обектът;
  - Третата помощна променлива Speed служи за изчисляване на скоростта на завъртане. Използва се формулата за скорост  $V = S/T$ ;
    - RotationOffset.Euler() превръща обектът Rotator(RotationOffset) в променлива с плаваща запетая - ъгъл на Ойлер, в градуси;
    - .Length() - взима дължината на преобразуваната променлива.
  - Последната помощна променлива е NewRotation от тип FRotator. RInterpConstantTo(Current, Target, DeltaTime, Speed) е метод на класът FMath, който служи за интерполиране от текущата ротация до желаната ротация непрекъснато.
  - Задаване на новата ротация с помощта на метода SetActorRelativeRotation.
- false - нищо не се случва.

```
void URotatorActor::SetShouldRotate(bool NewShouldRotate)
{
    ShouldRotate = NewShouldRotate;
}
```

Метод, който служи за задаване на променливата ShouldRotate.

## Scene Components

### Grabber

▼ Engine - Input

Input settings, including default input action and axis bindings.

These settings are saved in DefaultInput.ini, which is currently writable.

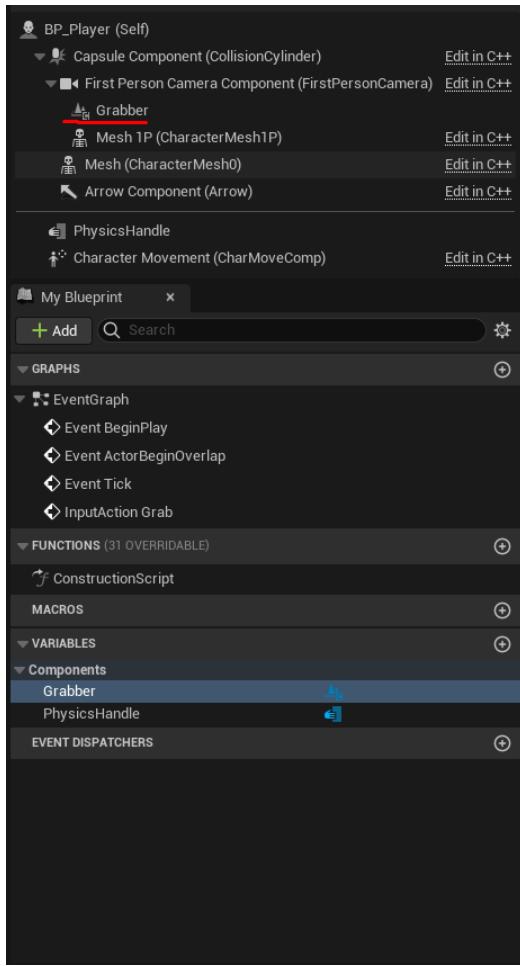
▼ Bindings

Speech Mappings 0 Array e

Action and Axis Mappings provide a mechanism to conveniently map keys and axes to input behaviors by inserting a layer of indirection between the key or axis and the final behavior. This allows for continuous range.

▼ Action Mappings + ⌛

Action	Input Type	Value	Shift	Ctrl	Alt	Cmd	⌘
Jump	Space Bar	Space Bar	Shift	Ctrl	Alt	Cmd	⌘
	Gamepad Face Button Bottom	Gamepad Face Button Bottom	Shift	Ctrl	Alt	Cmd	⌘
Grab	Left Mouse Button	Left Mouse Button	Shift	Ctrl	Alt	Cmd	⌘
	Gamepad Right Trigger	Gamepad Right Trigger	Shift	Ctrl	Alt	Cmd	⌘



Grabber.h

Променливи:

- float MaxGrabDistance – UPROPERTY, стойност, която определя на какво максимално разстояние може да се „хване“ обектът. Задава се в сантиметри, тъй като UE5 работи със сантиметри по подразбиране;
- float GrabRadius – UPROPERTY, използва се за задаване на радиуса на сферата, която се използва за Sweep;
- float HoldDostance – UPROPERTY, на какво разстояние от играла ще се намира обекта след „хвашане“;
- UPhysicsHandleComponent\* GetPhysicsHandle() const - метод от класа UPhysicsHandleComponent, който връща обект от този тип
  - UPhysicsHandleComponent - компонент, който служи за местене на физически обекти;

- #include "PhysicsEngine/PhysicalHandleComponent.h";
- bool GetGrabbableInReach(FHitResult& OutHitResult) const - проверява дали има обект за хващане;

Методи:

- void Release() - UNFUNCTION, използва се, когато се пусне обектът;
- void Grab() - UFUNCTION, използва се, когато се хване обектът.

## Grabber.cpp

```
UPhysicsHandleComponent* UGrabber::GetPhysicsHandle() const
{
    UPhysicsHandleComponent* Result = GetOwner() -> FindComponentByClass<UPhysicsHandleComponent>();
    if (Result == nullptr)
    {
        UE_LOG(LogTemp, Error, TEXT("Grabber requires a UPhysicsHandleComponent."));
    }
    return Result;
}
```

Най-напред се създава обект/указател Result от тип UPhysicsHandleComponent - текущият обект PhysicsHandle, който служи за местенето на физическите обекти в играта. FindComponentByClass намира компонент от съответният клас на текущия обект.

Проверява се дали има реален компонент след дефинирането.

- true - методът връща резултат - Result;
- false - изписва се съобщения на конзолата.

```

void UGrabber::Grab()
{
    UPhysicsHandleComponent* PhysicsHandle = GetPhysicsHandle();
    if(PhysicsHandle == nullptr)
    {
        return;
    }

    FHitResult HitResult;
    bool HasHit = GetGrabbableInReach(HitResult);
    if (HasHit)
    {
        UPrimitiveComponent* HitComponent = HitResult.GetComponent();
        HitComponent -> SetSimulatePhysics(true);
        HitComponent -> WakeAllRigidBodies();
        AActor* HitActor = HitResult.GetActor();
        HitActor -> Tags.Add("Grabbed");
        HitActor -> DetachFromActor(FDetachmentTransformRules::KeepWorldTransform);
        PhysicsHandle -> GrabComponentAtLocationWithRotation(
            HitComponent,
            NAME_None,
            HitResult.ImpactPoint,
            GetComponentRotation()
        );
    }
}

```

Създава се обект/указател от тип UPhysicsHandleComponent, който ще държи резултата от GetPhysicsHandle() метода - текущият обект PhysicsHandle, който служи за местенето на физическите обекти в играта. За да се избегнат грешки, се проверява дали такъв обект съществува. Ако да - се продължава нормално, ако не - функцията спира.

FHitResult HitResult обектът държи информация свързана с Line Tracing и Sweeping. Подава се като аргумент на метода GetGrabbableInReach и резултатът се запазва в булевата променлива HasHit.

Проверява се дали HitResult е true, с други думи дали пред героя стои обект, който е засечен от HitResult:

- true:
  - Създава се помощна променлива/указател HitComponent от тип UPrimitiveComponent и в него се записва информация за обектът, който е бил засечен от HitResult;
    - Primitive Components – scene components, които съдържат или генерираят някакъв вид геометрия, която обикновено служи за рендериране или се използва за collision;
  - Simulate Physics се задава като true на обектът, за да може да бъде хванат от героя
  - Извиква се WakeAllRigidBodies() - събужда се обектът и може да се хване свободно;
    - Rigid Body - обект, който може да симулира физика;
    - заради производителността, компоненти, които не са били хванати или манипулирани по някакъв начин от героят за известно време “заспиват”, което ги прави по-трудни за хващане ;
  - създава се променлива HitActor от тип AActor, който ще вземе информация за същият обект, който е засечен от HitResult, но използва Actor класа;
  - задава се таг, който се използва в друг клас и метод за проверка на тагове
  - DetachFromActor(FDetachmentTransformRules::KeepWorldTransform) - “отвързва” актьора (компонента) от всякакви други Scene компоненти, за които може да е бил прихванат;
    - FDetachmentTransformRules::KeepWorldTransform - запазва трансформациите на обекта според света в играта;
  - за хващането на обекта се използва GrabComponentAtLocationWithRotation методът. Аргументите са:
    - HitComponent;
    - NAME\_None - име на компонента, но се използва този синтаксис, когато не се подава име;
    - HitResult.ImpactPoint - местоположението в света на действителният контакта на trace/sweep формата с удареният обект;
    - GetComponentRotation() - ротацията на обекта,

```

bool UGrabber::GetGrabbableInReach(FHitResult& OutHitResult) const
{
    FVector Start = GetComponentLocation();
    FVector End = Start + GetForwardVector() * MaxGrabDistance;
    DrawDebugLine(GetWorld(), Start, End, FColor::Red);
    DrawDebugSphere(GetWorld(), End, 10, 10, FColor::Blue, false, 5);

    FCollisionShape Sphere = FCollisionShape::MakeSphere(GrabRadius);

    return GetWorld() -> SweepSingleByChannel(
        OutHitResult,
        Start,
        End,
        FQuat::Identity,
        ECC_GameTraceChannel2,
        Sphere);
}

```

Методът `GetGrabbableInReach` извършва sweep, който проверява дали по време на sweeping засича обекти на сцената. Това се прави с помощта на метода `SweepSingleByChannel`, който приема като аргументи:

- struct FHitResult& OutHit;
- const FVector& Start;
- const FVector& End;
- const FQuat& Rot;
- ECollisionChannel TraceChannel;
- const FCollisionShape7 CollisionShape;
- const FCollisionQueryParams& Params;
- const FCollisionResponseParams& ResponseParams.

За този проект са използвани следните аргументи:

- struct FHitResult& OutHit - променлива от тип FHitResult;
- const FVector& Start - вектор, на който стойността са първоначалните координати на компонента ;
  - Start = GetComponentLocation();
- const FVector& End - вектор, на който стойността му е резултат от следният израз:
  - End = Start + GetForwardVector() \* MaxGrabDistance;
- const FQuat& Rot - ротация на компонента;
  - FQuat::Identity - без ротация;
- ECollisionChannel TraceChannel - намира се като се отвори DefaultEngine.ini файлът от папката на проекта и се потърси името на класа Grabber. Там излиза информация в коя Trace Channel се намира този кас;
  - ECC\_GameTraceChannel12;
- const FCollisionShape7 CollisionShape - фигурата на sweep. Най-често се използва сфера;
  - FCollisionShape Sphere = FCollisionShape::MakeSphere(GrabRadius).

DrawDebugLine и DrawDebugSphere визуализират къде на екрана се извършва sweep и къде попада сферата след sweep

- #include “DrawDebugHelpers.h”.

```
void UGrabber::Release()
{
    UPhysicsHandleComponent* PhysicsHandle = GetPhysicsHandle();

    if (PhysicsHandle && PhysicsHandle -> GetGrabbedComponent())
    {
        AActor* GrabbedActor = PhysicsHandle -> GetGrabbedComponent() -> GetOwner();
        GrabbedActor -> Tags.Remove("Grabbed");
        PhysicsHandle -> ReleaseComponent();
    }
}
```

- отново се създава

PhysicsHandle обект, който помага за местенето на обектите.

Проверява се дали такъв обект съществува и дали има компонент, прихванат за него.

Ако да - взима се актьорът(обектът) и чрезdereferencing се взима притежателят на компонента. С други думи Root component.

Премахва се тагът, който е бил създаден в Grab() метода.

За пускането на обекта се използва ReleaseComponent() метода на UPhysicsHandleComponent класа.

## CollisionComponents

### TriggerComponent

TriggerComponent е клас, който наследява от BoxCollision.



TriggerComponent.h

Променливи:

- FName AcceptableActorTag – UPROPERTY, използва се за проверка на таговете на акторите в сценатаа. Те се задават директно през editor-а;
- AActor\* GetAcceptableActor() const - намира и връща правилният актор (обект) на сцената според зададения таг;
- UMover\* Mover - указател към класа Mover;
  - класа Mover трябва да се добави в текущия файл - #include "Mover.h".

Методи:

- void SetMover(UMover\* Mover) - UFUNCTION, инициализира Mover променливата (задава се през Blueprint).

TriggerComponent.cpp

```
void UTriggerComponent::SetMover(UMover* NewMover)
{
    Mover = NewMover;
}
```

Задаване на Mover

```

AAActor* UTriggerComponent::GetAcceptableActor() const
{
    TArray<AAActor*> Actors;
    GetOverlappingActors(Actors);

    for(AAActor* Actor : Actors)
    {
        bool HasAcceptableTag = Actor->ActorHasTag(AcceptableActorTag);
        bool IsGrabbed = Actor->ActorHasTag("Grabbed");
        if(HasAcceptableTag && !IsGrabbed)
        {
            return Actor;
        }
    }
    return nullptr;
}

```

Създава се масив от тип TArray, който ще съдържа актьори. С помощта на метода GetOverlappingActors се взимат актьорите, които се застъпват с Trigger компонента на сцената.

Във for цикъла се взимат индивидуалните актьори, които са в масива:

- създава се помощна булева променлива HasAcceptableTag, която съдържа резултат от проверката на ActorHasTag - дали е зададен съответстващ таг на текущият актьор от масива;
- създава се помощна булева променлива IsGrabbed, която съдържа резултат от проверката на ActorHasTag - дали има таг Grabbed, който се задава в класа Grabber;
- Проверява се дали и двете стойности са true:
  - true - метода връща актьора;
  - false - метода връща nullptr.

```

void UTriggerComponent::TickComponent(float DeltaTime, ELevelTick TickType, FActorComponentTickFunction* ThisTickFunction)
{
    Super::TickComponent(DeltaTime, TickType, ThisTickFunction);

    AAActor* Actor = GetAcceptableActor();
    if(Actor != nullptr)
    {
        UPrimitiveComponent* Component = Cast<UPrimitiveComponent>(Actor->GetRootComponent());
        if(Component != nullptr)
        {
            Component->SetSimulatePhysics(false);
        }
        Actor->AttachToComponent(this, FAttachmentTransformRules::KeepWorldTransform);
        Mover->SetShouldMove(true);
    }

    else
    {
        Mover->SetShouldMove(false);
    }
}

```

В TickComponent метода, който се извиква на всеки фрейм в играта, се създава обект Actor от тип AAActor\*, който държи резултат от метода GetAcceptableActor().

Проверява се дали обектът не е nullptr:

- true:
  - извършва се каст, който преобръща актьора в UPrimitiveComponent
    - не е сигурно дали Root компонента на актьора е от тип UPrimitiveComponent;
    - прави се проверка, дали не съществува такъв компонент:
      - true - задаваме SimulatePhysics на false;
      - false - нищо не се случва.
  - Actor -> AttachToComponent(this, FAttachmentTrasnformRules::KeepWorldTRansform)
    - методът прави така, че актьорът, който е хванат в момента се захваща за актьора, към когото принадлежи Trigger компонента;
  - Mover -> SetShouldMove(true) - задава се променливата ShoudMove на класа Mover на true.
- false:
  - Mover -> SetShouldMove(false) - задава се променливата ShoudMove на класа Mover на false.

## RotatorComponent

RotatorComponent е клас, който наследява от BoxCollision.



## RotatorComponent.h

Променливи:

- FName AcceptableActorTag – UPROPERTY, използва се за проверка на таговете на актьорите в сцената. Тя се задават директно през editor-а;
- AActor\* GetAcceptableActor() const - намира и връща правилният актьор (обект) на сцената според зададения таг;
- URotatorActor\* Rot - указател към класа RotatorActor;
  - класа RotatorActor трябва да се добави в текущия файл - #include "RotatorActor.h".

Методи:

- SetRotator(URotatorActor\* RotatorActor) - UNFUNCTION, инициализира RotatorActor променливата (задава се през Blueprint).

### RotatorComponent.cpp

```
void URotatorComponent::SetRotator(URotatorActor* NewRotator)
{
    Rot = NewRotator;
}
```

Задаване на Rotator

```
AActor* URotatorComponent::GetAcceptableActor() const
{
    TArray<AActor*> Actors;
    GetOverlappingActors(Actors);

    for(AActor* Actor : Actors)
    {
        bool HasAcceptableTag = Actor->ActorHasTag(AcceptableActorTag);
        bool IsGrabbed = Actor->ActorHasTag("Grabbed");
        if(HasAcceptableTag && !IsGrabbed)
        {
            return Actor;
        }
    }
    return nullptr;
}
```

Създава се масив от тип TArray, който ще съдържа актьори. С помощта на метода GetOverlappingActors се взимат актьорите, които се застъпват с Trigger компонента на сцената.

Във for цикъла се взимат индивидуалните актьори, които са в масива:

- създава се помощна булева променлива HasAcceptableTag, която съдържа резултат от проверката на ActorHasTag - дали е зададен съответстващ таг на текущият актьор от масива;

- създава се помощна булева променлива IsGrabbed, която съдържа резултат от проверката на ActorHasTag - дали има таг Grabbed, който се задава в класа Grabber;
- Проверява се дали и двете стойности са true:
  - true - метода връща актьора;
  - false - метода връща nullptr.

```
void URotatorComponent::TickComponent(float DeltaTime, ELevelTick TickType, FActorComponentTickFunction* ThisTickFunction)
{
    Super::TickComponent(DeltaTime, TickType, ThisTickFunction);

    AActor* Actor = GetAcceptableActor();
    if(Actor != nullptr)
    {
        UPrimitiveComponent* Component = Cast<UPrimitiveComponent>(Actor->GetRootComponent());
        if(Component != nullptr)
        {
            Component->SetSimulatePhysics(false);
        }
        Actor->AttachToComponent(this, FAttachmentTransformRules::KeepWorldTransform);
        Rot->SetShouldRotate(true);
    }
    else
    {
        Rot->SetShouldRotate(false);
    }
}
```

- в

TickComponent метода, който се извиква на всеки фрейм в играта, се създава обект Actor от тип AActor\*, който държи резултат от метода GetAcceptableActor().

Проверява се дали обектът не е nullptr:

- true:
  - извършва се каст, който преобръща актьора в UPrimitiveComponent;
    - не е сигурно дали Root компонента на актьора е от тип UPrimitiveComponent;
    - прави се проверка, дали не съществува такъв компонент:
      - true - задаваме SimulatePhysics на false;
      - false - нищо не се случва.
  - Actor->AttachToComponent(this, FAttachmentTrasnformRules::KeepWorldTTransform) - методът прави така, че актьорът, който е хванат в момента се захваща за актьора, към когото принадлежи Trigger компонента;
  - Rot-> SetShouldRotate(true) - задава се променливата ShoudRotate на класа RotatorActor на true.
- false:
  - Rot-> SetShouldRotate(false) - дава се променливата ShoudRotate на класа RotatorActor на false.

## Blueprint

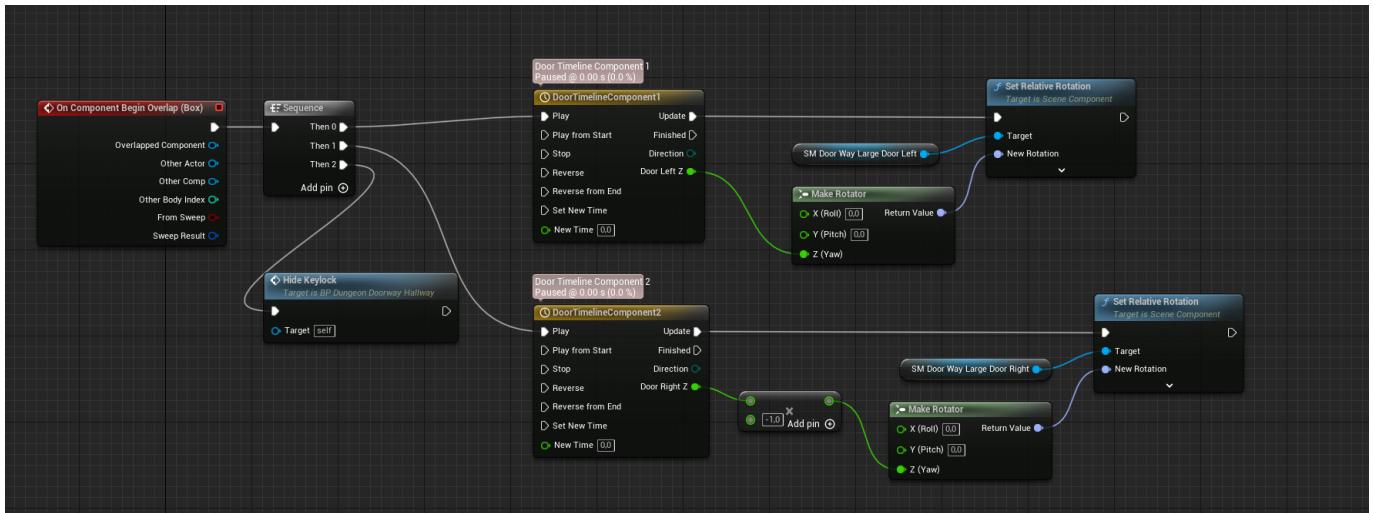
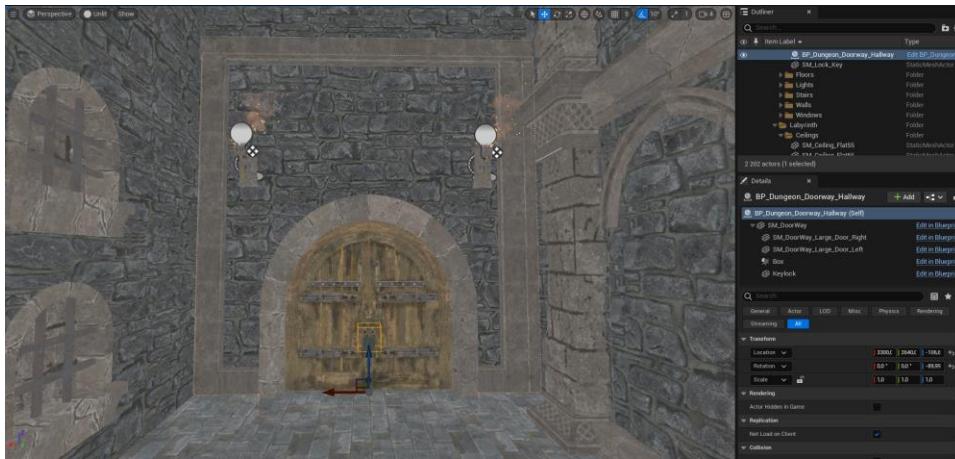
Използвани класове и методи:

- On Component Begin Overlap;
- Create Widget;
- Setters ;
- Getters;
- Add to Viewport;
- Remove all Widgets;
- Timelines;
- Make Rotator;
- Set Relative Rotation;
- Custom Hide Function;
- Widgets:
  - Notes;
  - Buttons.
- Get Player Controller;
- Enable Input;
- Get All Actors of Class;
- Custom Open Door function;
- Set Text;
- Set Input Mode Game And UI;
- Set Input Mode Game Only;
- Flow Control Nodes:
  - Branch;
  - Sequence.

### Механизъм за отваряне на вратите

Използва се за всички врати, които използват Blueprint код вместо C++ за отварянето им. Самата функция, която отваря вратите е подобна навсякъде. Разликата е, че има зададени условия при някои врати, например проверка на променливи или набиране на специфичен код преди отваряне.

Hallway Doors

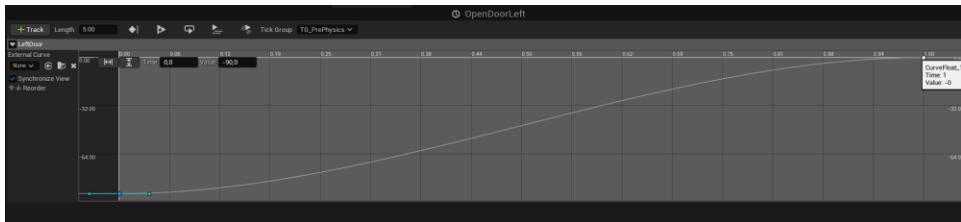


On Component Begin Overlap е вид Event node. Използва се в Blueprint за да дефинира и засече събитие в играта. Event nodes са началната точка на Blueprint скриптове. On Begin Overlap засича кога два обекта се припокриват в следствие на което изпълняват някакъв скрипт. В случая вратите трябва да се отворят.

Sequence е вид механизъм - Flow Control, който контролира в каква последователност ще се изпълнят последващите функции. В този скрипт трябва вратите да се отворят едновременно след припокриване, което е именно функцията на Sequence - трите функции след Sequence ще се изпълнят едновременно.

Timeline nodes са специални възли, които предоставят проектиране на анимации, базирани на време и изпълняването им.

- Play – Timeline-a се възпроизвежда напред от текущото си време;
- Reverse - възпроизвежда се обратно от текущото си време;
- Door Left/Right Z- предоставя променлива с плаваща запетая, която съдържа информация за анимацията - време и стойност.

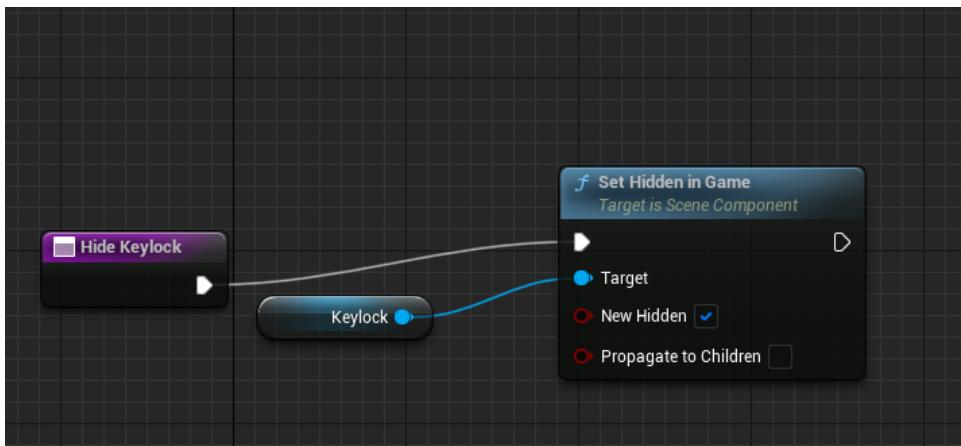


Използва се Timeline Curve за да се опише анимацията на движението. Тъй като вратите трябва да се отворят до 90 градуса, първата точка, която в случая е началната позиция на вратата, се задава да е 0, а втората - 90. Същото се прави и за вторият компонент на вратата.

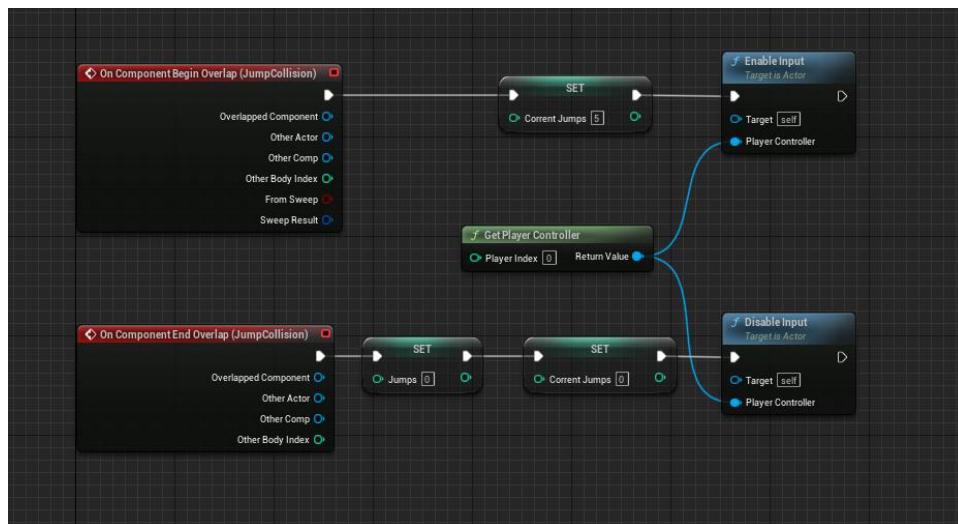
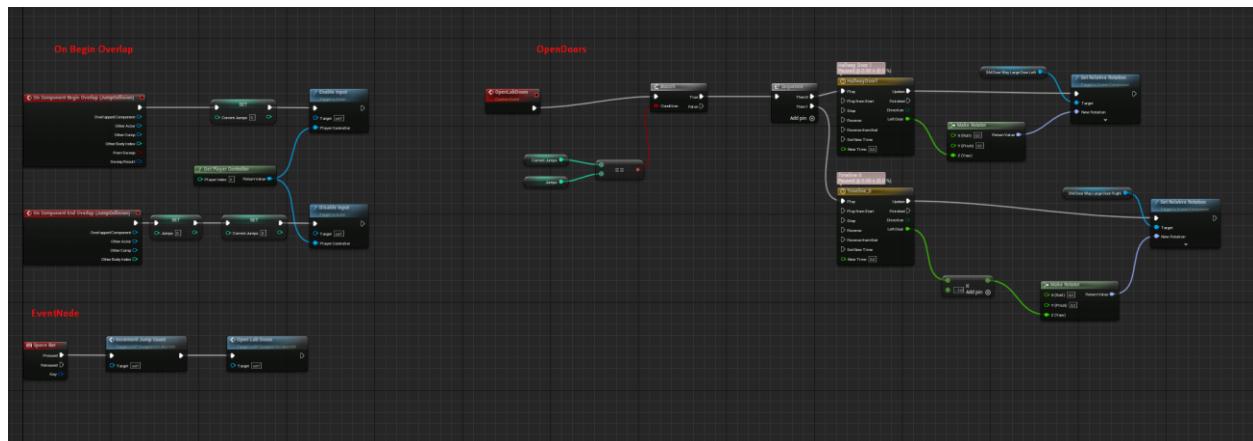
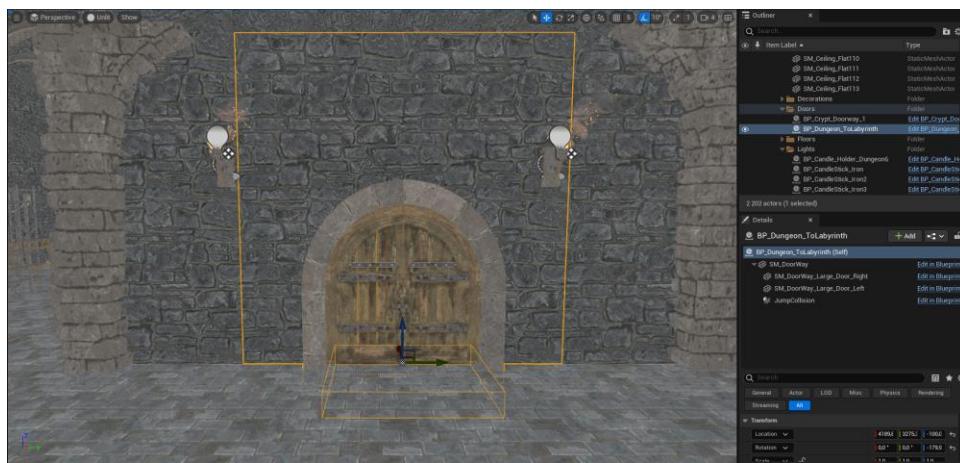
Door Left/Right Z изходите се свързват с метод Make Rotator - той прави Rotator обект, който се използва за самата ротация на вратата. Свързването става чрез Z(Yaw), тъй като той отговаря за ротацията по Z.

- В случаите на десния компонент, стойността на Door Right Z трябва да се умножи по -1, защото тя се отваря в обратната посока.

За задаване на новата ротация се използва функцията Set Relative Rotation, която за Target приема компонента, който трябва да бъде завъртан, а за New Rotation се приема стойността, която връща Make Rotator



Hide Keylock е новосъздадена функция, която скрива компонента катинар, след отваряне на вратите. За Target приема референция към компонента Keylock. New Hidden се задава на true директно от възела, който е атрибут на компонента. Тази функция ще се изпълни веднага, щом настъпи при покриването на обектите и вратите започнат да се отварят.



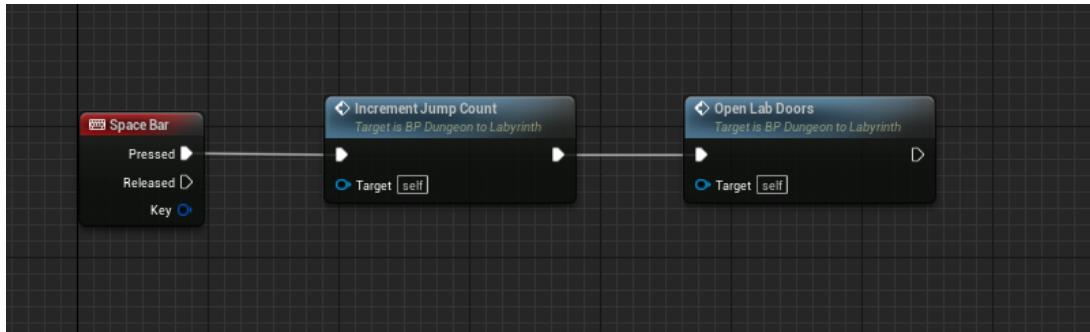
Когато настъпи припокриване на обектите се задава стойността на **Correct Jumps** с помощта на **Setter**.

**Enable Input** позволява въвеждане от клавиатурата.

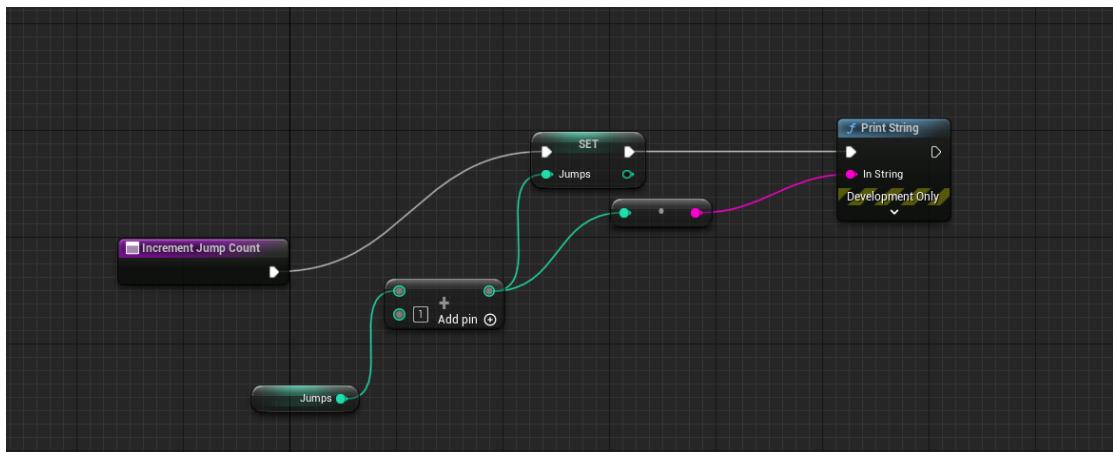
Get Player Controller връща контролера на текущия играч. Подава се на Enable Input

В On Component End Overlap стойността на променливата Jumps, която брои колко пъти е бил натиснат бутона Space на клавиатурата, се задава обратно на 0, за да може логиката да е правилна.

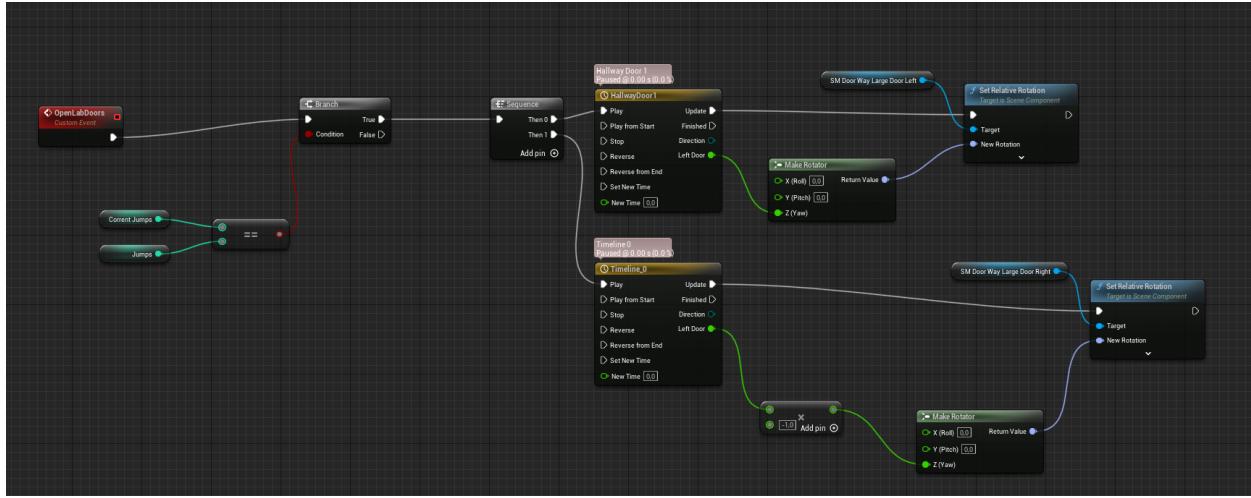
Disable Input забранява въвеждането от клавиатурата, което настъпва, след като двата обекта спрат да се припокриват.



Space Bar е Event node, който се извика всяки път, когато бутоњът бъде натиснат. След всяко натискане на бутона се изпълнява функцията Increment Jump Count. След Increment Jump Count се изпълнява Open Lab Doors.



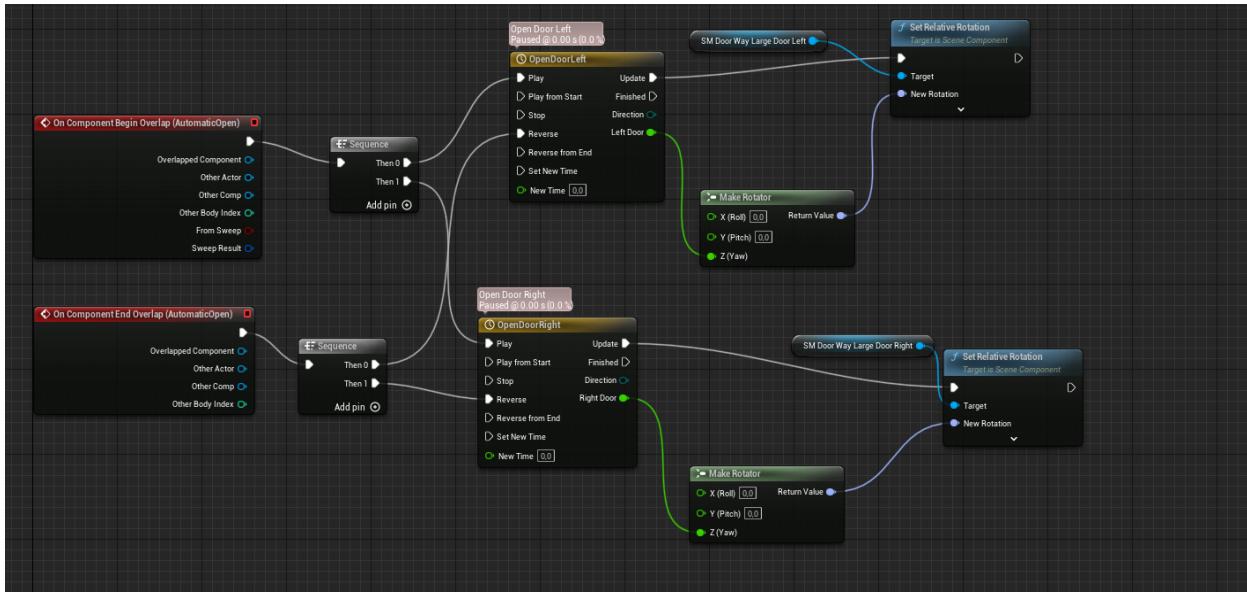
Функцията Increment Jump Count прави просто изчисление. Всеки път когато бъде натиснат съответният бутоң, стойността на Jumps се увеличава с 1. За да се види каква е стойността на променливата се използва Print String, който принтира на екрана стойността.



В този скрипт, механизмът, който отваря вратите е поставен в Custom Event Node, за по-лесна четимост. Използван е Branch, който е Blueprint еквивалента на if в C++. Прави се проверка дали двете променливи са еднакви, защото от това зависи дали вратите ще се отворят или не. Условието е изнесено е отделен възел.

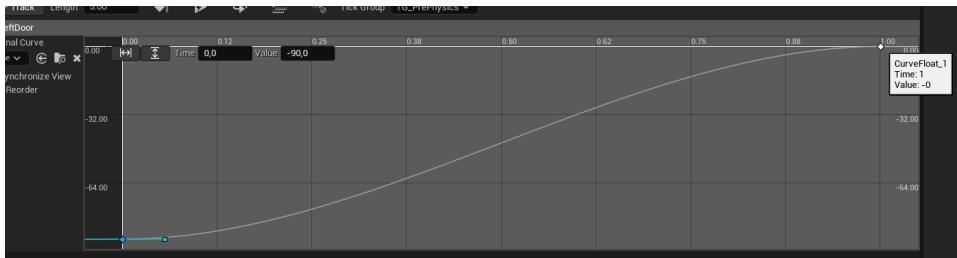
## Labyrinth Doors 2



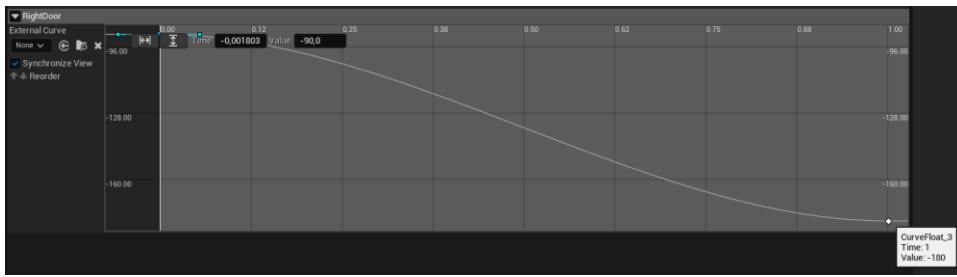


Използван е Reverse, което прави анимацията на обратно. С други думи, която при покриването или колизията приключи, анимацията се обръща и вратите се затварят. При повторна поява на колизията вратите отново се отварят и съответно затварят.

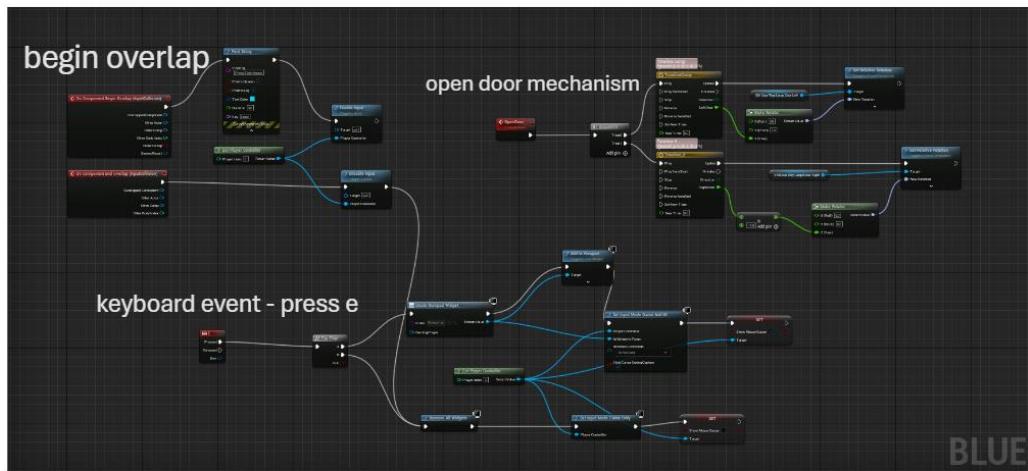
Тук умножението по -1 преди създаването на Rotator е изпуснато, тъй като оригиналната ротация на вратите не започва от 0. Зададени са директни стойности в timeline, с което се избягва умножението:

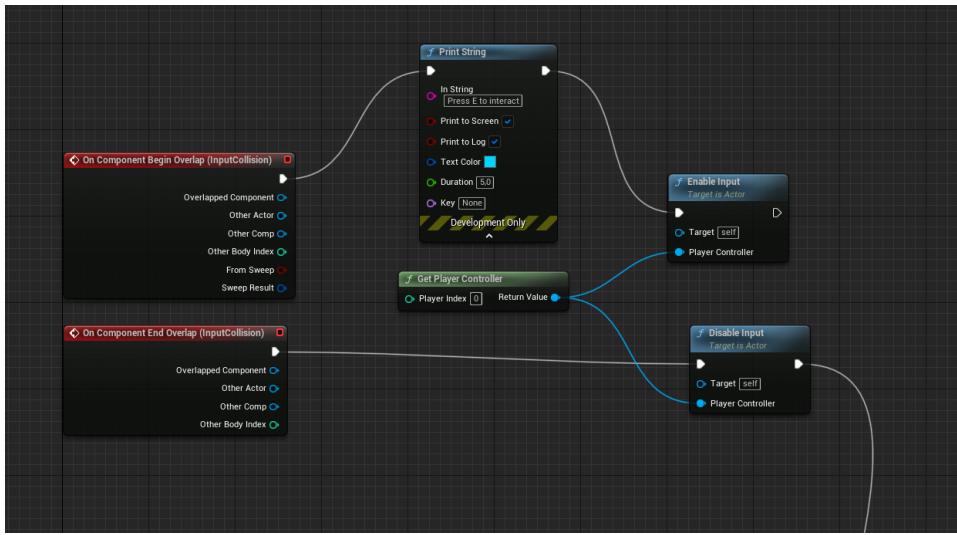


Ротацията на лявата врата започва от -90 градуса и стига до 0 градуса.

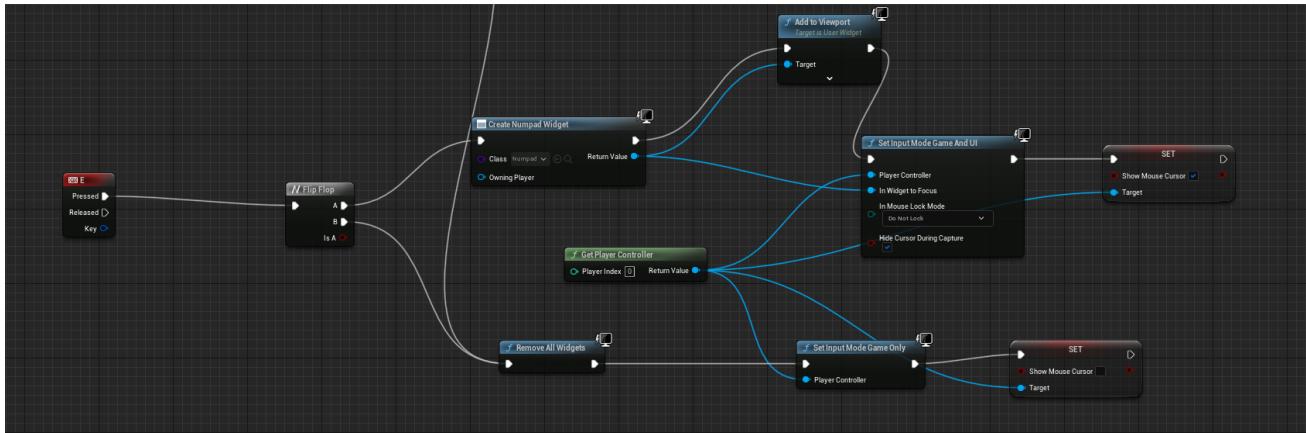


## Small Room Doors





В Begin Overlap новото е, че се принтира на екрана текст, който казва на играта как се активира панелът с бутоните. Бутонът "E" се използва в този случай.



След първо натискане на "E" се създава панел - Widget с помощта на функцията Create Numpad Widget. Като class се избира направеният предварително панел в играта.

Add to Viewport се използва за визуализиране на панела на екрана.

Set Input Mode Game And UI се използва за настройване на режим на въвеждане, който позволява на потребителският интерфейс да отговаря на въвеждането от потребителя и ако потребителският интерфейс не го обработва, играчът или контролерът на играта получава шанс за обработване.

- Player Controller взима стойност от Get Player Controller функцията;
- In Widget to Focus се подава връщаната стойност от функцията Create Numpad Widget.

Set се използва за сътвание на изгледа. Подава се Player Controller като Target и Show Mouse Cursor се задава на true, за да се показва мишката на екрана.

FlipFlop служи за превключване между двата изхода за изпълнение.

След повторно натискане на бутона “E” панелът се скрива от екрана и играчът може да продължи да се движи из нивото.

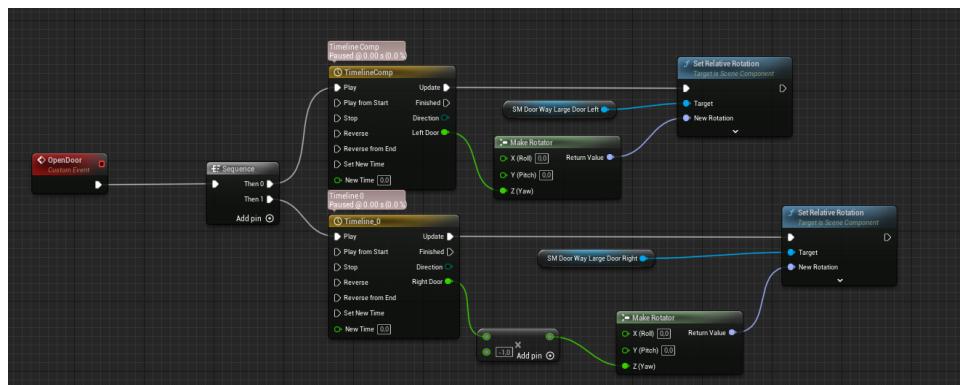
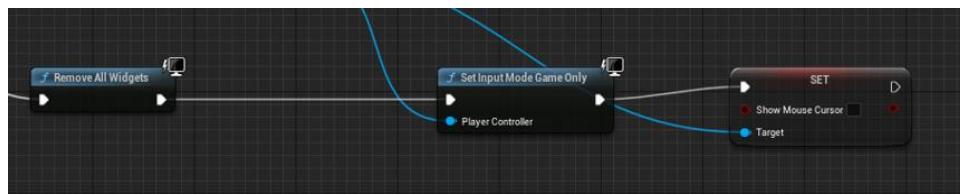
Remove All Widget премахва всички панели, които до момента са видими на екрана.

Set Input Mode Game Only настройва режим на въвеждане, който позволява само въвеждането от играча или контролера да отговаря на въвеждането от потребителя. С други думи въвеждането от потребителският интерфейс се забранява.

- Player Controller отново се подава като резултат от функцията Get Player Controller.

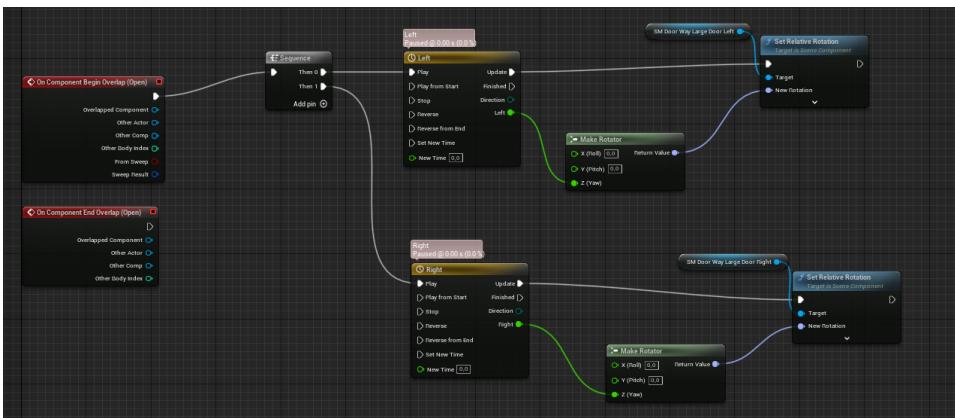
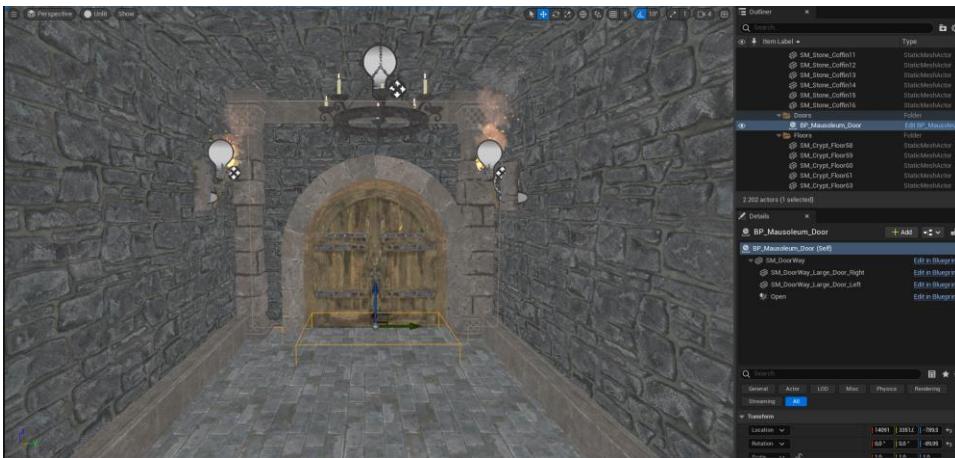
Извиква се Set, който премахва мишката от екрана.

Disable Input се използва за скриване на панела с бутоните и е свързан директно със скрипта, който се изпълнява след повторно натискане на бутона “E”, защото логиката е същата.

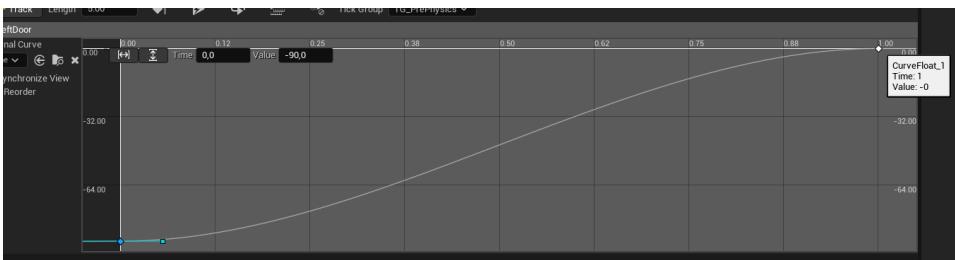


OpenDoor е изнесен в отделен Event Node за по-добра четимост и използваемост. Логиката за отваряне на вратите е същата.

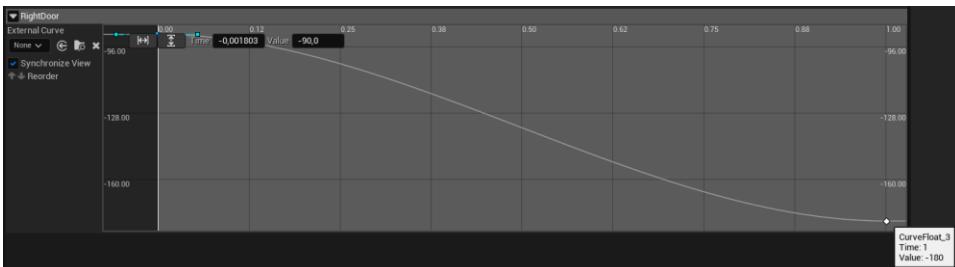
## Mausoleum Doors



Отново умножението по  $-1$  е изпуснато, тъй като оригиналната ротация на вратите не започва от  $0$ . Зададени са директни стойности в timeline, с което се избягва умножението:



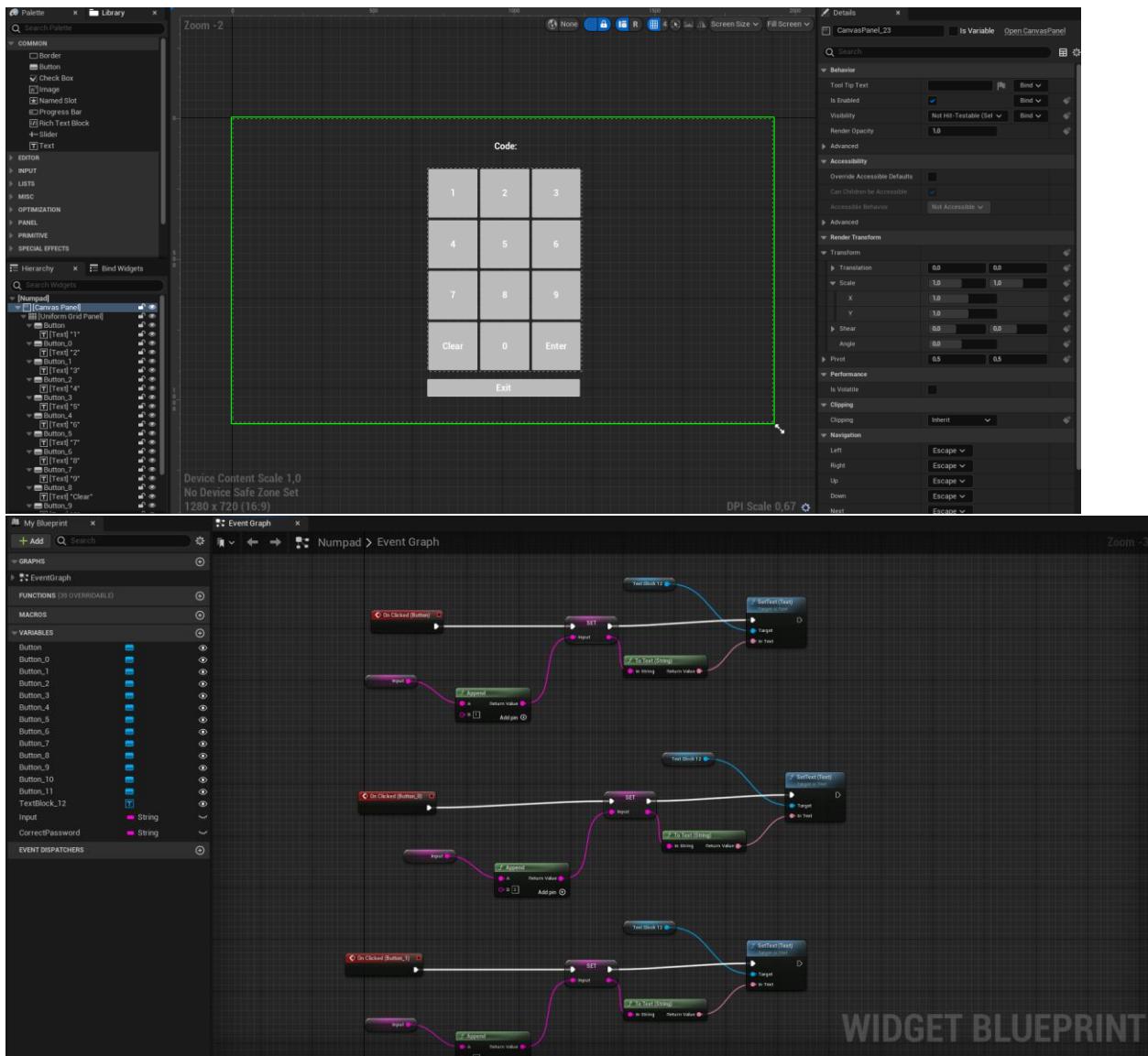
Ротацията на лявата врата започва от  $-90$  градуса и стига до  $0$  градуса.



Ротацията на дясната врата започва от  $-90$  градуса и стига до  $-180$  градуса.

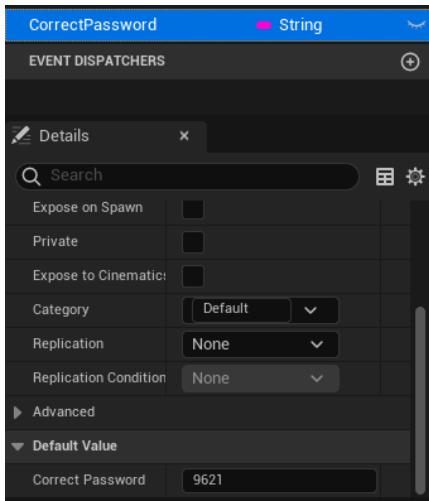
# Widgets

## Small Room Door Numpad



В Blueprint скрипта на панела се добавя Event Node за всеки бутон с числов стойност.

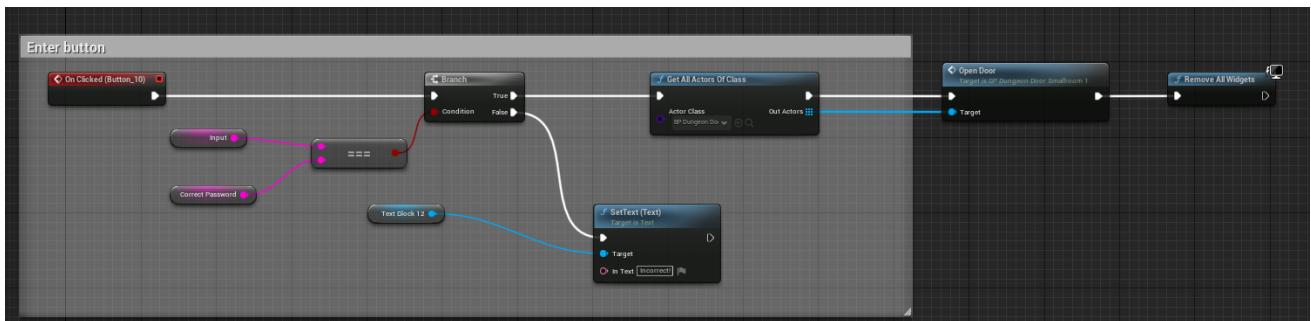
Променливата Input държи в себе си кода, който се набира от играча, а CorrectPassword е правилният код, който трябва да се набере.



След натискане на всеки един бутон, с помощта на Append функцията се добавя стойноста, на която отговаря бутонът. След задаване на стойността се извика функцията SetText, която

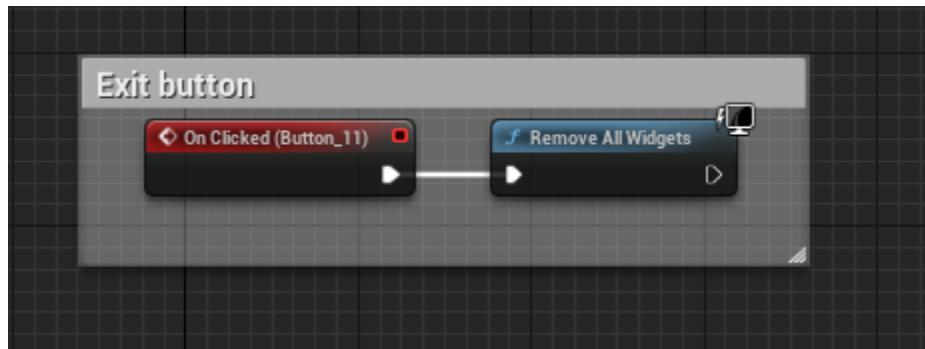
За всеки един от бутоните 0-9 се случва един и същи процес - при натискане на бутона:

- Извиква се Setter на променливата Input и чрез функцията Append, се добавя числото, на съответният бутон, който е натиснат;
- Резултата от Set се подава на To Text функцията, която обръща променливата от тип String към тип Text;
- Извиква се SetText. Резултата от To Text се подава в InText, а Target е Text Block 12, което в панела е текста, който изписва набраната комбинация от числа над бутоните:

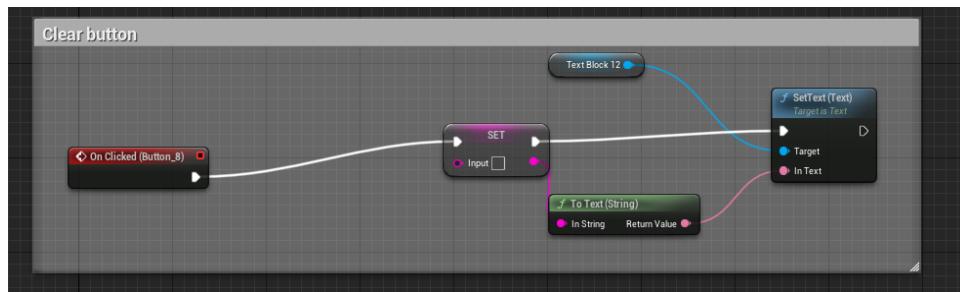


Enter бутона служи за потвърждаване на паролата/кода. Отново се използва On Clicked event node. Branch служи за проверка на въведенния код от играча и правилния код.

- true:
  - Извиква се функцията Get All Actors Of Class, който връща всички актьори, които принадлежат на заданият клас, в случая BP\_DungeonDoor;
  - След това се извиква custom event node-a Open Door, в който се намира стандартната логика за отваряне на вратите.
- false:
  - Над бутоните се изписва, че паролата е грешна.

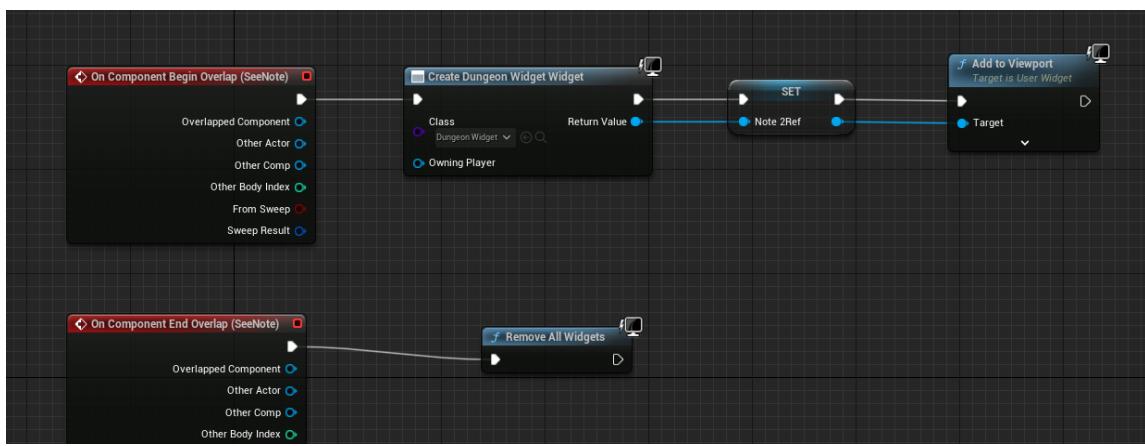
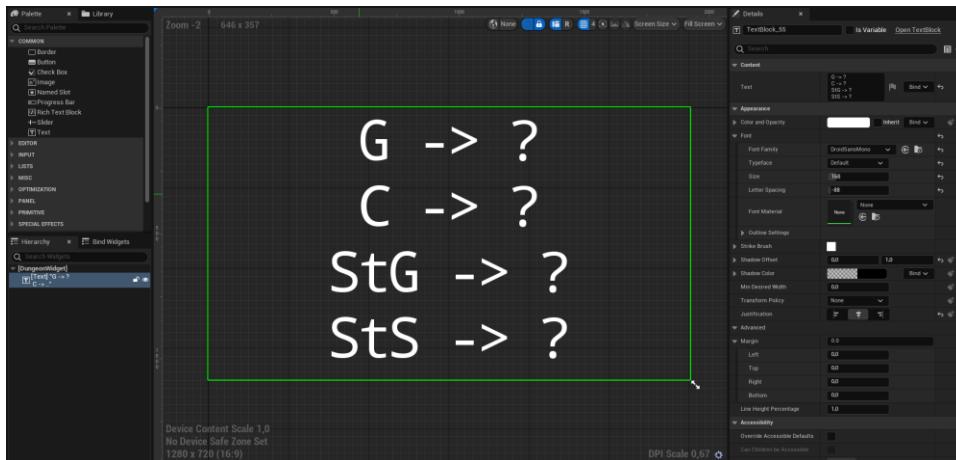


Exit бутона служи за премахване на панела от екрана.



Clear бутона служи за изчистване на набраната към момента парола. Чрез Set се задава празен String и се подава към полето над бутоните, т.е. изчезва въведената парола.

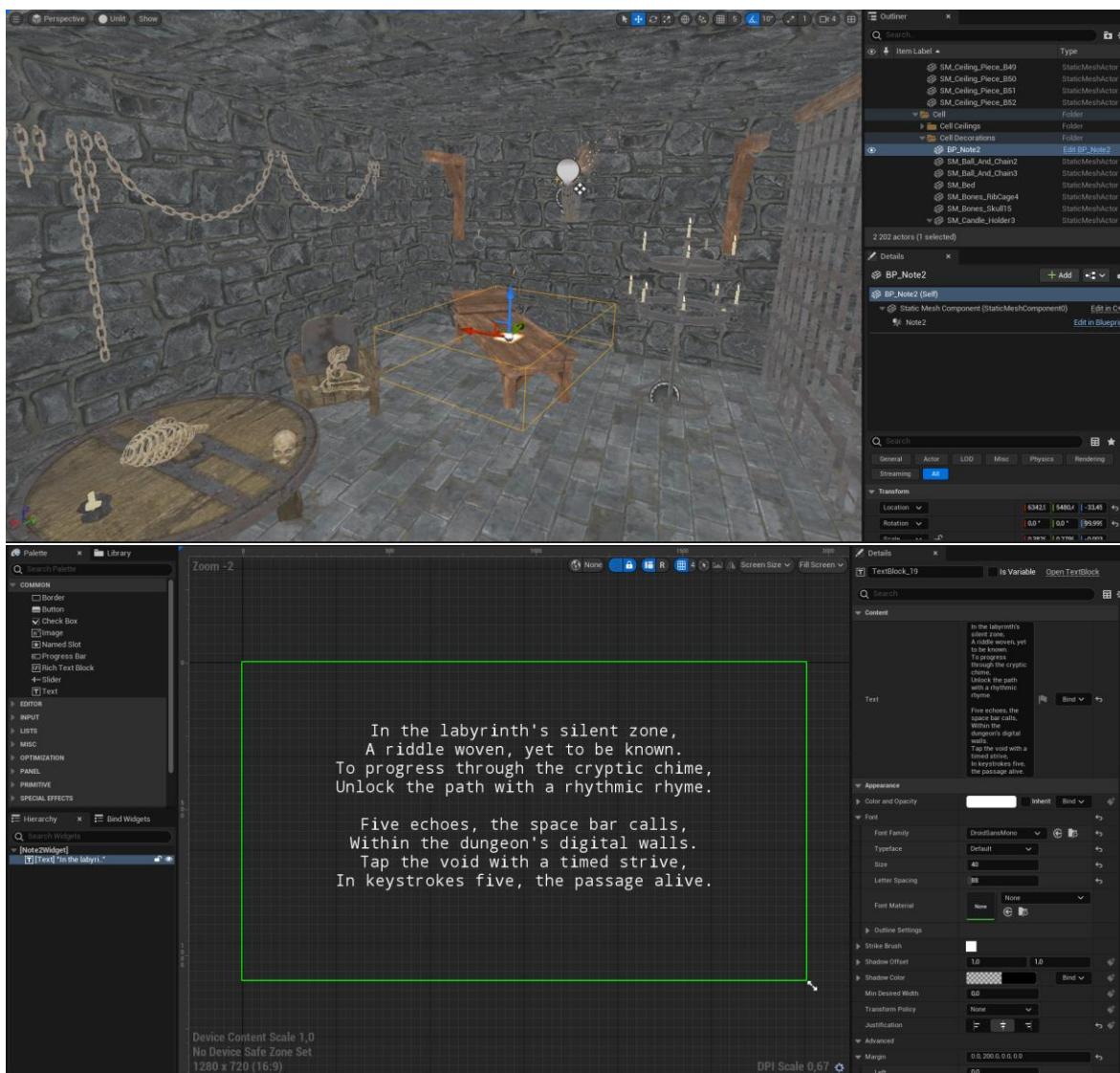
Small Room Hint Note

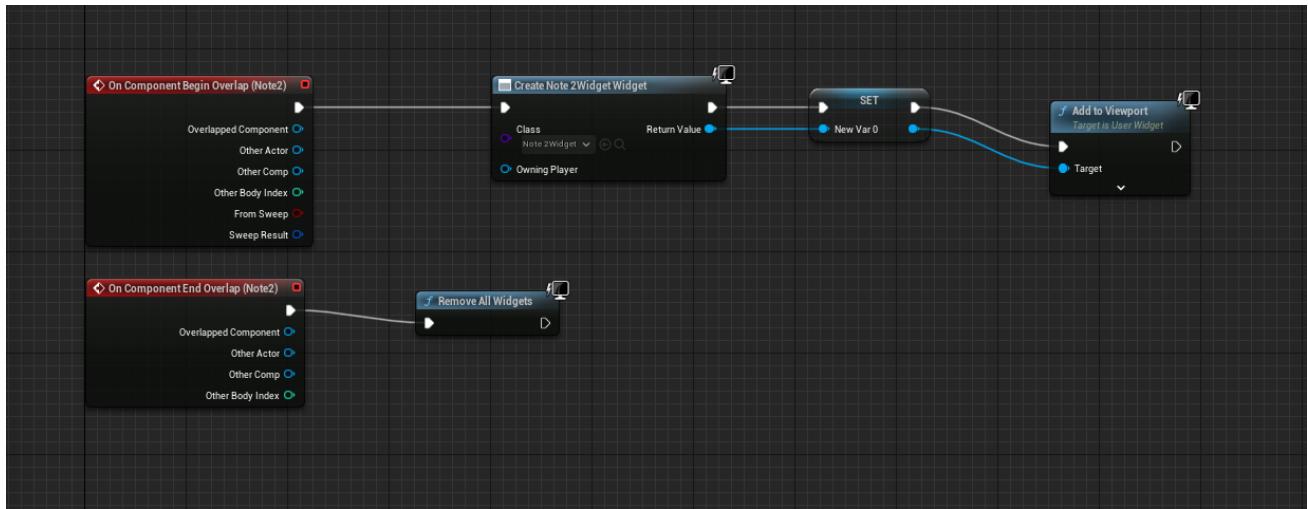


Стандартно, при припокриване на обекти (при всичките подсказки е нужно играчът да стъпи в полето, зададено за козилия) се активира Event node-а On Component Begin Overlap. Създава се панел от вече съществуващ такъв с името Dungeon Widget, който съдържа текст с подсказка. Добавя се към изгледа.

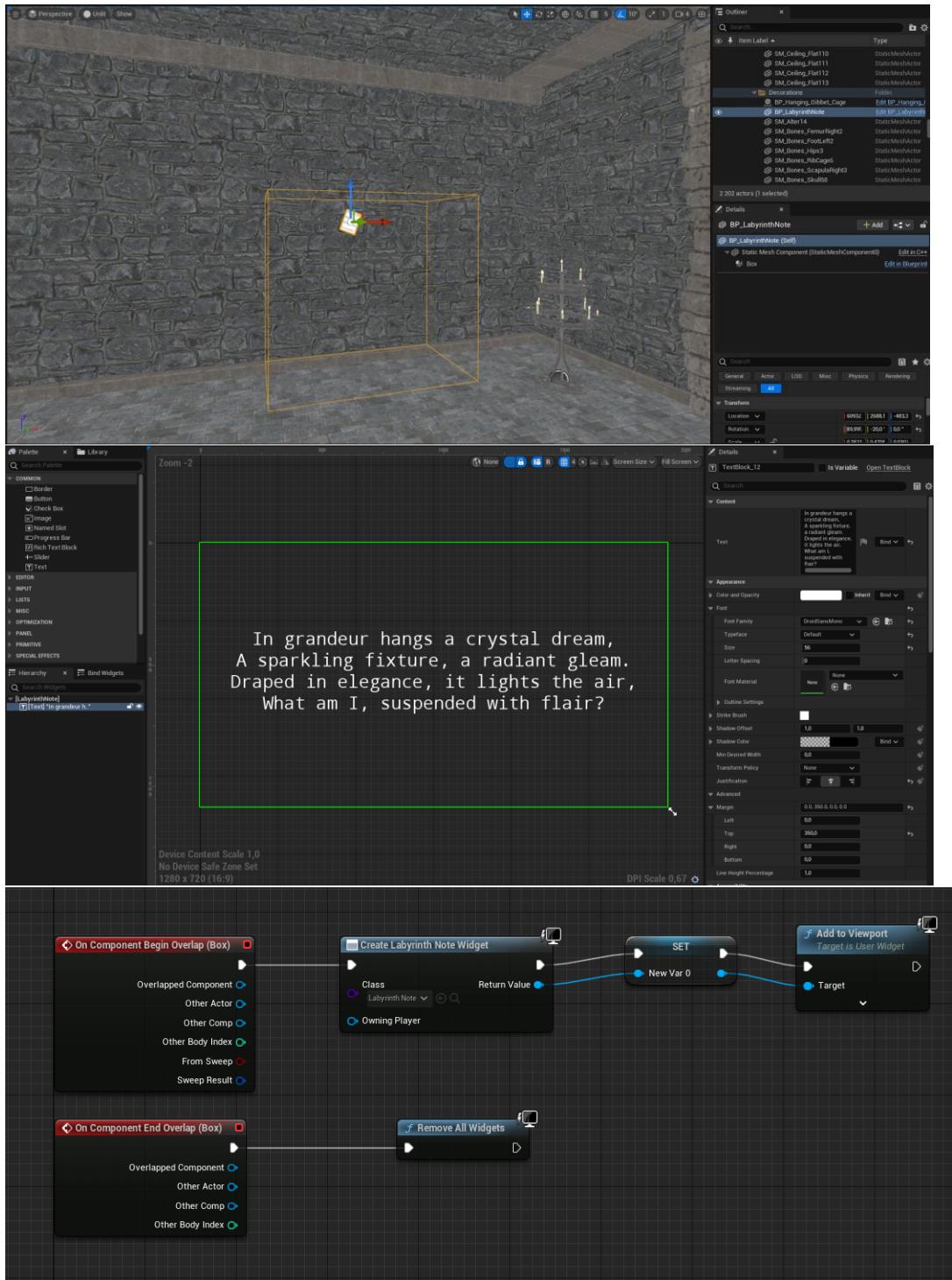
При прекъсване на припокриването или колизията се премахва панела с помощта на Remove All Widgets.

### Cell Hint Note

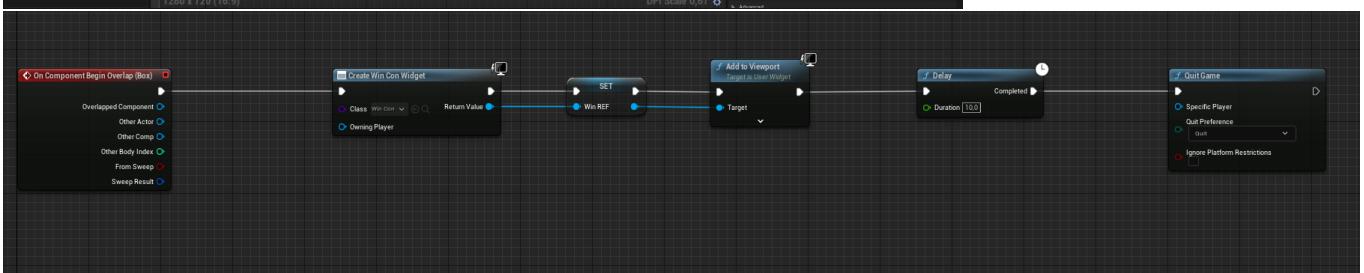
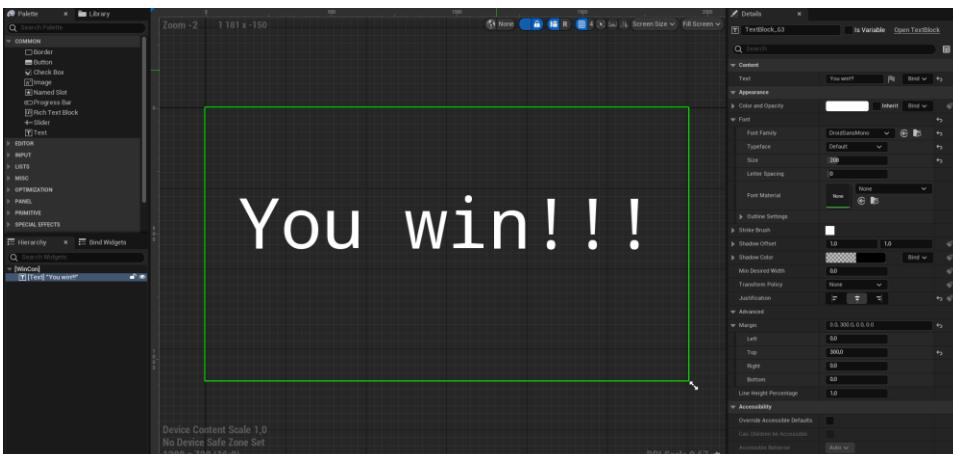




Labyrinth Hint Note 1



## Mausoleum



Стандартно, след припокриване се показва панелът на екрана.

Delay се използва за забавено извикване на следващата функция.

След като минат 10 секунди, играта спира автоматично с помощта на функцията Quit Game.