Whiskey Business/ HUD

Architecture/Design Document

Table of Contents

Contents

Change History

Version: 0.3

Modifier: Daniel Azevedo

Date: 04 / 02 / 2025

Description of Change: Setup hearts and have them appearing on screen

Version: 0.4

Modifier: Daniel Azevedo

Date: 17 / 02 / 2025

Description of Change: HUD elements can be moved on screen with toggleHUD()

Version: 0.5

Modifier: Daniel Azevedo

Date: 20 / 03 / 2025

Description of Change: HUD now gets created per player to be replicated.

Version: 0.6

Modifier: Daniel Azevedo

Date: 1 / 04 / 2025

Description of Change: HUD fully works with respawn and is fully replicated for each player.

1. Introduction

This document illustrates how the QuetzalHUD class and the custom Slate classes work together. This project uses Slate instead of UMG for the HUD elements to have more customizability within the code. The three custom Slate classes created were SHealthWidget, SScoreWidget, and STextWidget.

The HUD will work differently depending on the selected mode. For this build, the HUD will be in "Stock" mode, meaning that the score Widgets will be used to display each character's lives.

The SHealthWidget class creates Heart icons on the HUD that will be pushed back into a vector of hearts for each player. These will be modified to have hearts removed or have hearts cut in half correlating to the health of the player.

The SScoreWidget class creates Score/Stock icons on the HUD that will increase based on the score of each player. If the player's score increases past 5, the score icons will instead be replaced by a number counting the player's score instead.

The STextWidget class creates Text on the HUD that currently increases based on the score of the player and will be set up to count down the time for a game timer. The Text widget is currently not functional while in stock mode.

2. Design Goals

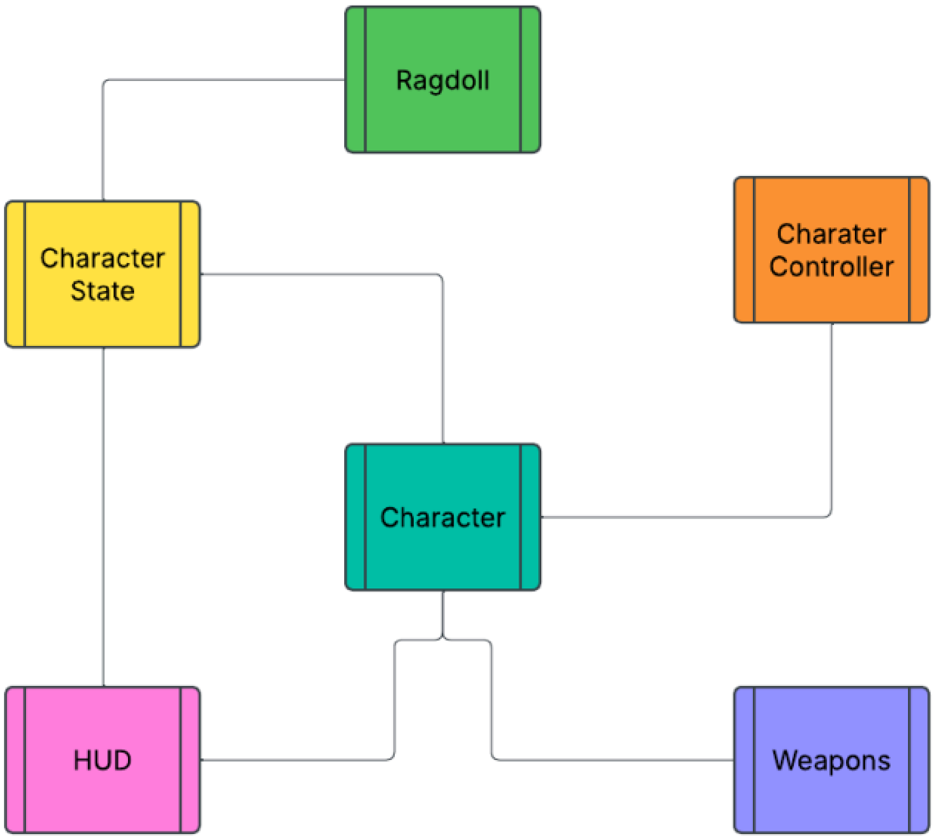
- **Ability to be toggled:** The HUD will have the ability to be toggled to allow the players to see the entire screen without any obstructions.
- **Not to clutter the screen:** When the HUD is on screen, it should be non-invasive and not block any gameplay elements.
- **Clear Designs:** Players should easily be able to determine what each HUD element represents upon seeing them.

3 System Behaviour

The HUD will be created by the GameMode and will have a pointer to the player's controller to get their state HP when health is updated.

4 Logical View

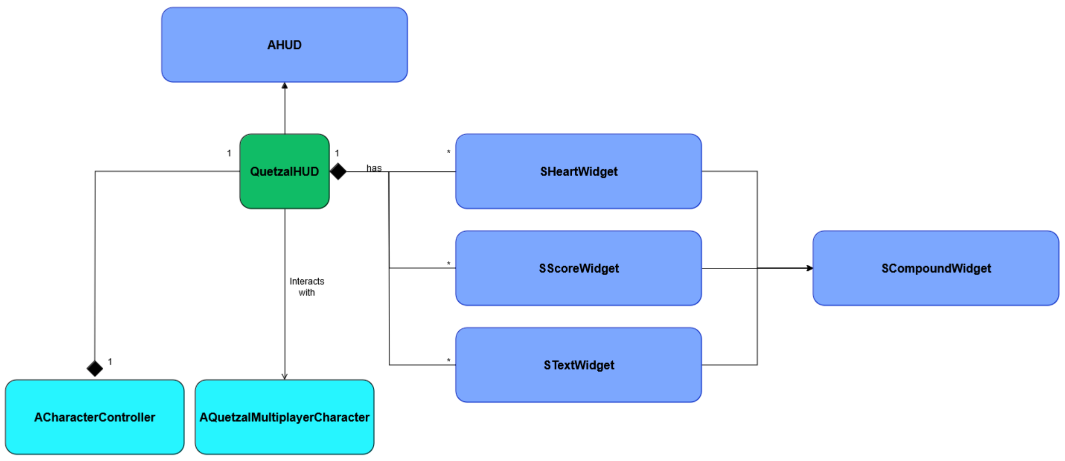
4.1 High-Level Design



High-Level Design of varying systems in the game.

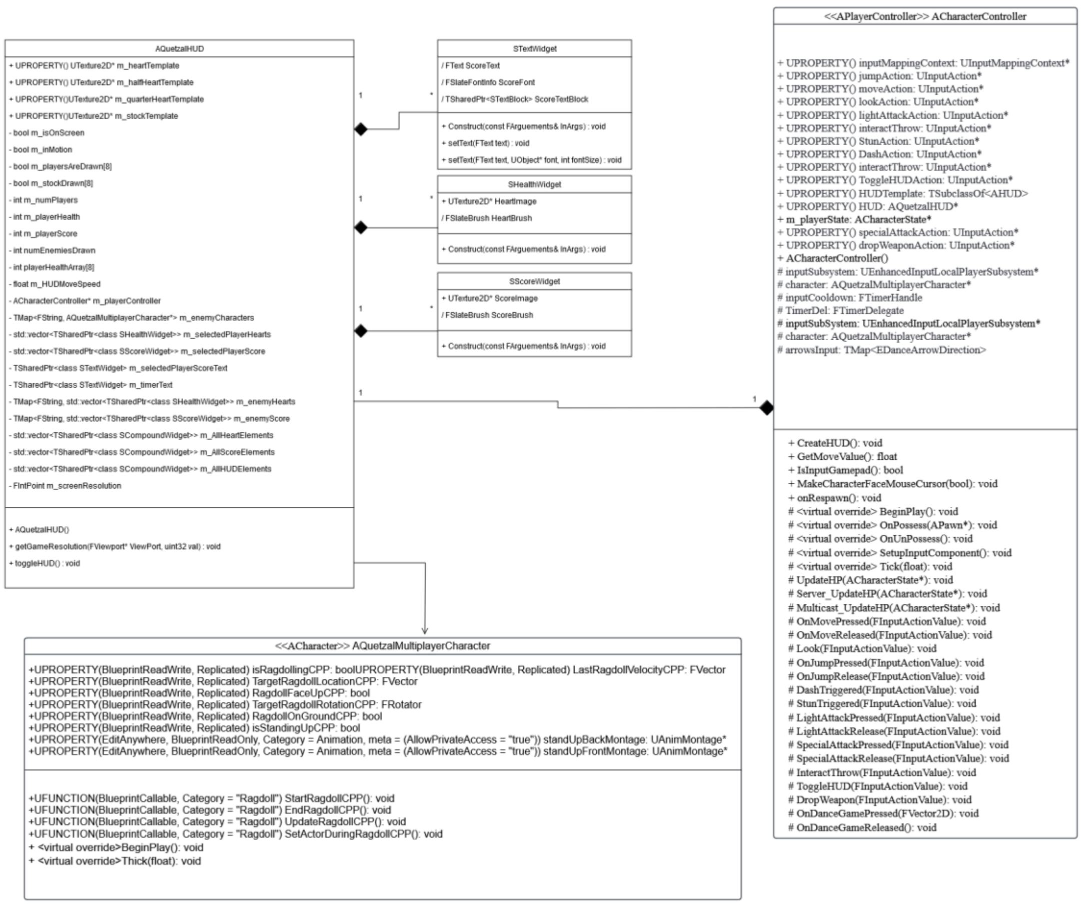
- The Character Controller will handle the input from the user that will control the player.
- The HUD will keep the players informed of the current health and score of the players using values from Character and Character State.
- Weapons will interact with the player by activating equipped weapons or receiving damage from weapons.
- Depending on the Character State the player can enter into a Ragdoll state.
- At the center is the Character interacting with all systems in some varying ways.

4.2 Mid-Level Design of the HUD



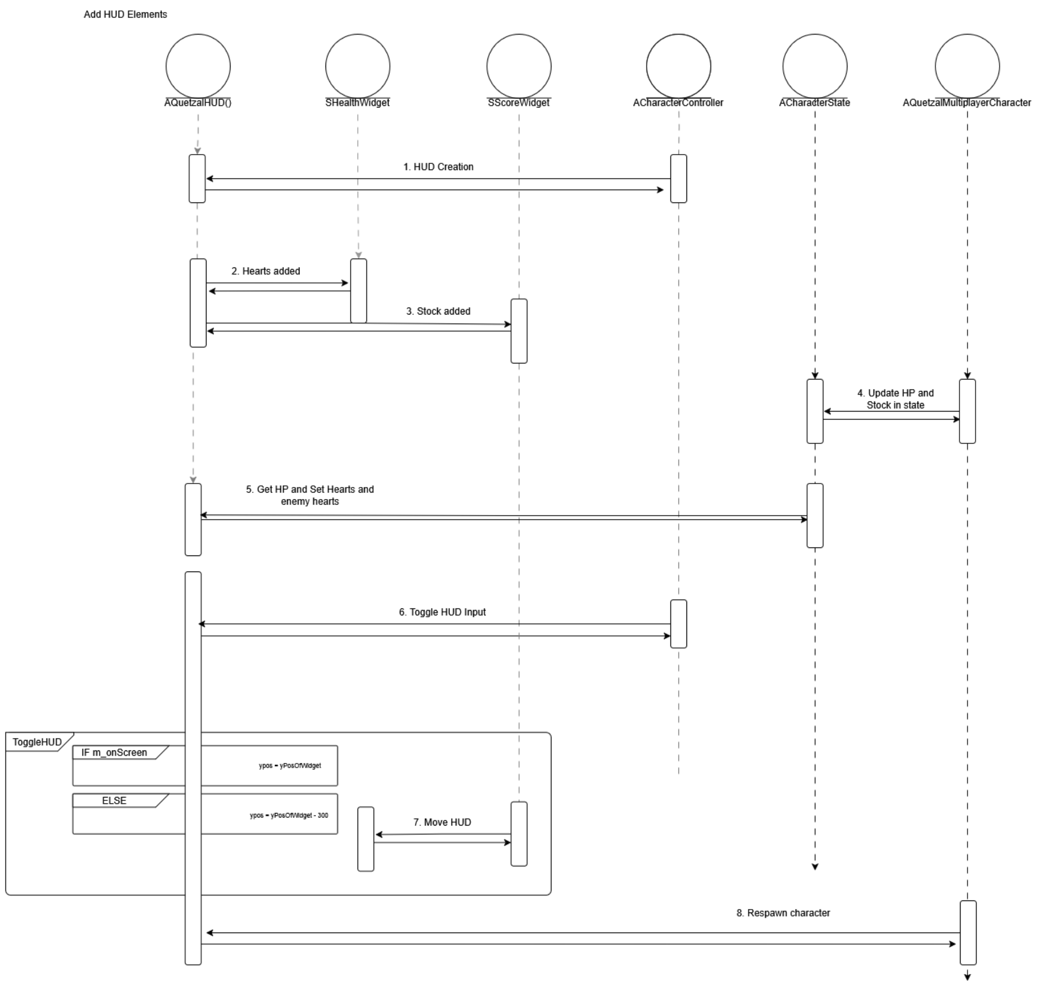
The HUD is spawned from the character controller and has 3 widgets that it uses to make the HUD elements. On creation the HUD will create its own players hearts, and their own stock. When other players join it will add their hearts and stock as well. It has interactions setup with the character to respawn the player and update the HUD when the player takes damage.

4.3 Detailed Class Design of the HUD


AQuetzalHUD
+ UPROPERTY() UTexture2D* m_heartTemplate
+ UPROPERTY() UTexture2D* m_halfHeartTemplate
+ UPROPERTY()UTexture2D* m_quarterHeartTemplate
+ UPROPERTY()UTexture2D* m_stockTemplate
- bool m_isOnScreen
- bool m_inMotion
- bool m_playersAreDrawn[8]
- bool m_stockDrawn[8]
- int m_numPlayers
- int m_playerHealth
- int m_playerScore
- int numEnemiesDrawn
- int playerHealthArray[8]
- float m_HUDMoveSpeed
- ACharacterController* m_playerController
- TMap<FString, AQuetzalMultiplayerCharacter*> m_enemyCharacters
- std::vector<TSharedPtr<class SHealthWidget>> m_selectedPlayerHearts
- std::vector<TSharedPtr<class SScoreWidget>> m_selectedPlayerScore
- TSharedPtr<class STextWidget> m_selectedPlayerScoreText
- TSharedPtr<class STextWidget> m_timerText
- TMap<FString, std::vector<TSharedPtr<class SHealthWidget>> m_enemyHearts
- TMap<FString, std::vector<TSharedPtr<class SScoreWidget>> m_enemyScore
- std::vector<TSharedPtr<class SCompoundWidget>> m_AllHeartElements
- std::vector<TSharedPtr<class SCompoundWidget>> m_AllScoreElements
- std::vector<TSharedPtr<class SCompoundWidget>> m_AllHUDElements
- FIntPoint m_screenResolution
+ AQuetzalHUD()
+ getGameResolution(FViewport* ViewPort, uint32 val) : void
+ toggleHUD() : void
1
*
STextWidget
/ FText ScoreText
/ FSlateFontInfo ScoreFont
/ TSharedPtr<STextBlock> ScoreTextBlock
+ Construct(const FArguements& InArgs) : void
+ setText(FText text) : void
+ setText(FText text, UObject* font, int fontSize) : void
1
*
SHealthWidget
+ UTexture2D* HeartImage
/ FSlateBrush HeartBrush
+ Construct(const FArguements& InArgs) : void
1
*
SScoreWidget
+ UTexture2D* ScoreImage
/ FSlateBrush ScoreBrush
+ Construct(const FArguements& InArgs) : void
1
1
<<APlayerController>> ACharacterController
+ UPROPERTY() inputMappingContext: UInputMappingContext*
+ UPROPERTY() jumpAction: UInputAction*
+ UPROPERTY() moveAction: UInputAction*
+ UPROPERTY() lookAction: UInputAction*
+ UPROPERTY() lightAttackAction: UInputAction*
+ UPROPERTY() interactThrow: UInputAction*
+ UPROPERTY() StunAction: UInputAction*
+ UPROPERTY() DashAction: UInputAction*
+ UPROPERTY() interactThrow: UInputAction*
+ UPROPERTY() ToggleHUDAction: UInputAction*
+ UPROPERTY() HUDTemplate: TSubclassOf<AHUD>
+ UPROPERTY() HUD: AQuetzalHUD*
+ m_playerState: ACharacterState*
+ UPROPERTY() specialAttackAction: UInputAction*
+ UPROPERTY() dropWeaponAction: UInputAction*
+ ACharacterController()
# inputSubsystem: UEnhancedInputLocalPlayerSubsystem*
# character: AQuetzalMultiplayerCharacter*
# inputCooldown: FTimerHandle
# TimerDel: FTimerDelegate
# inputSubSystem: UEnhancedInputLocalPlayerSubsystem*
# character: AQuetzalMultiplayerCharacter*
# arrowsInput: TMap<EDanceArrowDirection>
+ CreateHUD(): void
+ GetMoveValue(): float
+ IsInputGamepad(): bool
+ MakeCharacterFaceMouseCursor(bool): void
+ onRespawn(): void
# <virtual override> BeginPlay(): void
# <virtual override> OnPossess(APawn*): void
# <virtual override> OnUnPossess(): void
# <virtual override> SetupInputComponent(): void
# <virtual override> Tick(float): void
# UpdateHP(ACharacterState*): void
# Server_UpdateHP(ACharacterState*): void
# Multicast_UpdateHP(ACharacterState*): void
# OnMovePressed(FInputActionValue): void
# OnMoveReleased(FInputActionValue): void
# Look(FInputActionValue): void
# OnJumpPressed(FInputActionValue): void
# OnJumpRelease(FInputActionValue): void
# DashTriggered(FInputActionValue): void
# StunTriggered(FInputActionValue): void
# LightAttackPressed(FInputActionValue): void
# LightAttackRelease(FInputActionValue): void
# SpecialAttackPressed(FInputActionValue): void
# SpecialAttackRelease(FInputActionValue): void
# InteractThrow(FInputActionValue): void
# ToggleHUD(FInputActionValue): void
# DropWeapon(FInputActionValue): void
# OnDanceGamePressed(FVector2D): void
# OnDanceGameReleased(): void
<<ACharacter>> AQuetzalMultiplayerCharacter
+UPROPERTY(BlueprintReadWrite, Replicated) isRagdollingCPP: boolUPROPERTY(BlueprintReadWrite, Replicated) LastRagdollVelocityCPP: FVector
+UPROPERTY(BlueprintReadWrite, Replicated) TargetRagdollLocationCPP: FVector
+UPROPERTY(BlueprintReadWrite, Replicated) RagdollFaceUpCPP: bool
+UPROPERTY(BlueprintReadWrite, Replicated) TargetRagdollRotationCPP: FRotator
+UPROPERTY(BlueprintReadWrite, Replicated) RagdollOnGroundCPP: bool
+UPROPERTY(BlueprintReadWrite, Replicated) isStandingUpCPP: bool
+UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Animation, meta = (AllowPrivateAccess = "true")) standUpBackMontage: UAnimMontage*
+UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Animation, meta = (AllowPrivateAccess = "true")) standUpFrontMontage: UAnimMontage*
+UFUNCTION(BlueprintCallable, Category = "Ragdoll") StartRagdollCPP(): void
+UFUNCTION(BlueprintCallable, Category = "Ragdoll") EndRagdollCPP(): void
+UFUNCTION(BlueprintCallable, Category = "Ragdoll") UpdateRagdollCPP(): void
+UFUNCTION(BlueprintCallable, Category = "Ragdoll") SetActorDuringRagdollCPP(): void
+ <virtual override>BeginPlay(): void
+ <virtual override>Thick(float): void

5 Process View of the HUD

Add HUD Elements



The HUD elements are all made through Slate widgets created in AQuetzalHUD CreateHUD function.

Process Steps:

1. **HUD Creation**
   - When the character controller gets created, it will start a timer that after a few seconds, the HUD will get created

- The timer is used to ensure that the player state is initialized when the HUD is created.
- When created the HUD needs to take a pointer to the player controller, this will be used to get this player's PlayerState

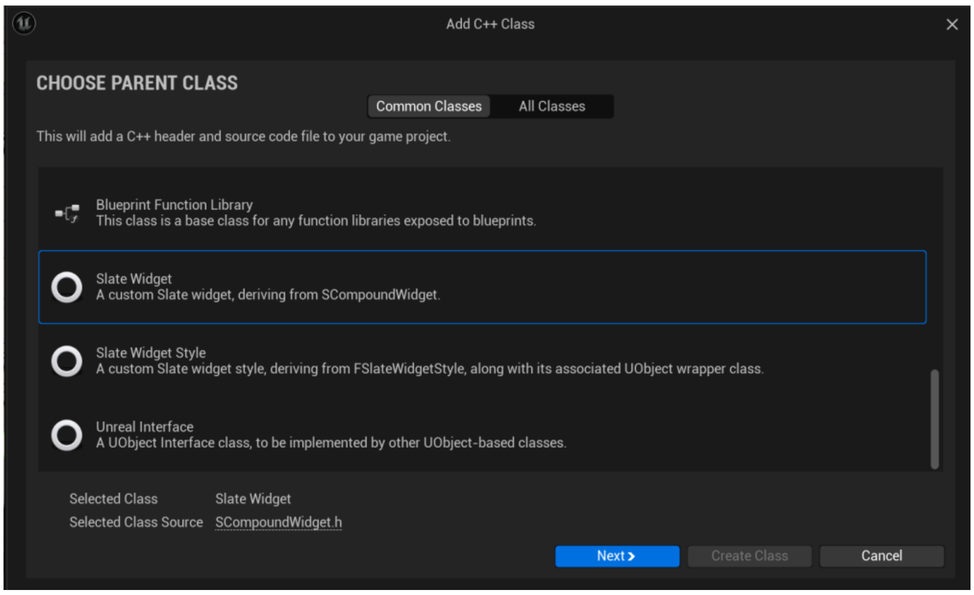
2. **Hearts Added**
    - When the player gets created, they will create their own hearts
    - As enemies join, their hearts will get added as well and the HUD will keep track of the enemies to not re-add them.
3. **Stock Added**
    - When the player gets created, they will create their own stock icons
    - As enemies join, their stock icons will get added as well and the HUD will keep track of the enemies to not re-add them.
4. **Update HP and Stock**
    - The player character updates the player's HP in the PlayerState (the HP in player state is replicated ensuring all players have an accurate HP value for each player)
5. **Get HP and Set Hearts and enemy Hearts**
    - For both the player and enemies, if their current HP is not equal to their last documented HP (which gets stored when the spawn and everytime they take damage) their hearts will be updated
    - This works for both healing and taking damage (but is not the way that players hearts are respawned
6. **Toggle HUD Input**
    - When the player clicks the ToggleHUD button, all HUD elements will be moved off screen or on screen depending on their current position.
    - This will toggle the m_onScreen variable
7. **Move HUD**
    - The HUD knows where to move the HUD elements to thanks to a HUD element struct made in QuetzalHUD.h.
    - The elements get added to a Vector of all the HUD elements that keeps track of their min and max positions to know where they need to get moved to and when to stop moving them.
8. **Respawn Character**

- When the owning player dies, the Respawn() function gets called to remove one of their stock icons and re-add their hearts
- When an enemy player dies, the addEnemyHearts() function gets called. This will remake the enemies hearts and if the enemy had already been added to the HUD, it will remove one of their Stock icons

6 Use Case View

If you wanted to create new screen widgets:

- Create a new C++ class with a SCompoundWidget parent



Add C++ Class

CHOOSE PARENT CLASS

This will add a C++ header and source code file to your game project.

Blueprint Function Library
This class is a base class for any function libraries exposed to blueprints.

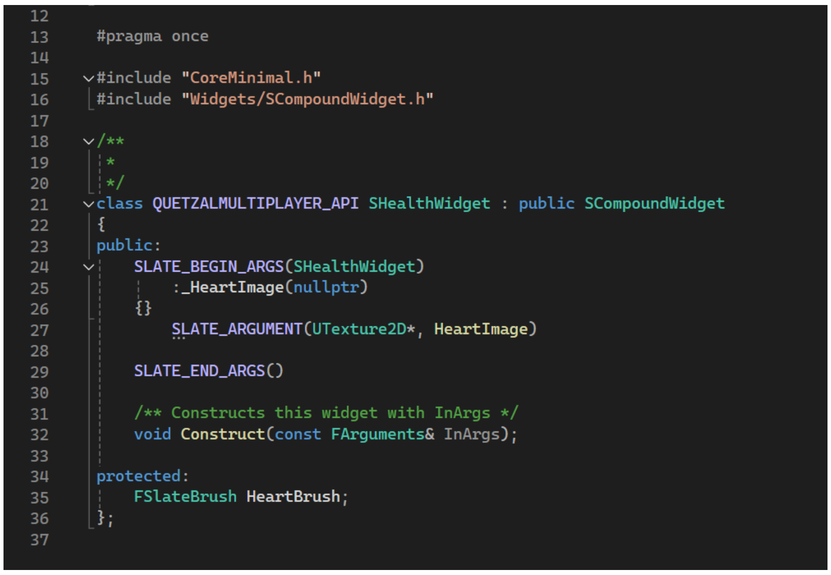
Slate Widget
A custom Slate widget, deriving from SCompoundWidget.

Slate Widget Style
A custom Slate widget style, deriving from FSlateWidgetStyle, along with its associated UObject wrapper class.

Unreal Interface
A UObject Interface class, to be implemented by other UObject-based classes.

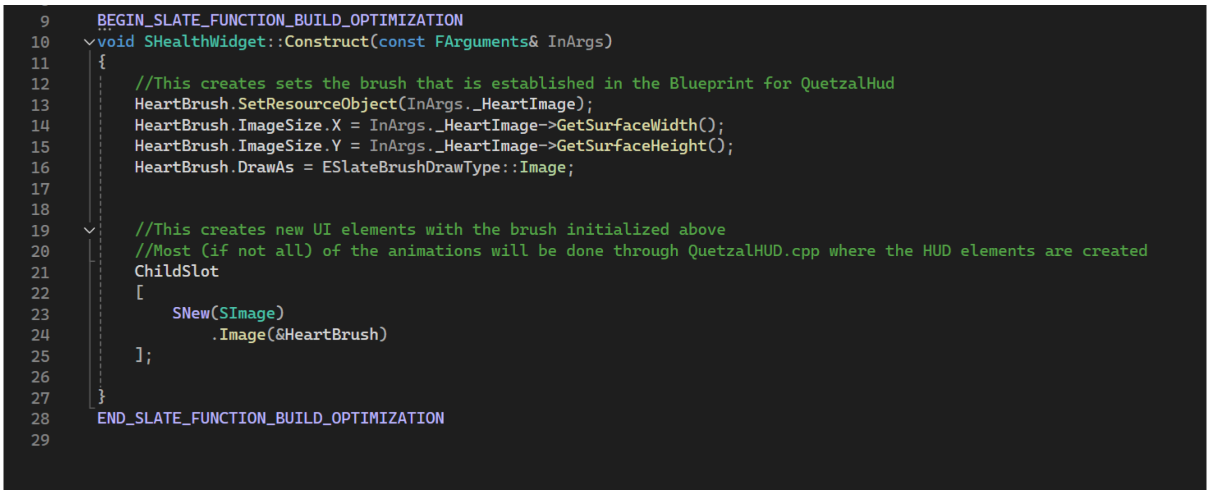
Selected Class: Slate Widget
Selected Class Source: SCompoundWidget.h

- Set up the .h file with all the variables you want (in this case we just want a brush to display the image on screen)
- Also set up the SLATE_ARGUEMENT to set what your Widget will take in when created (in this case we want to take in a texture to render on screen)



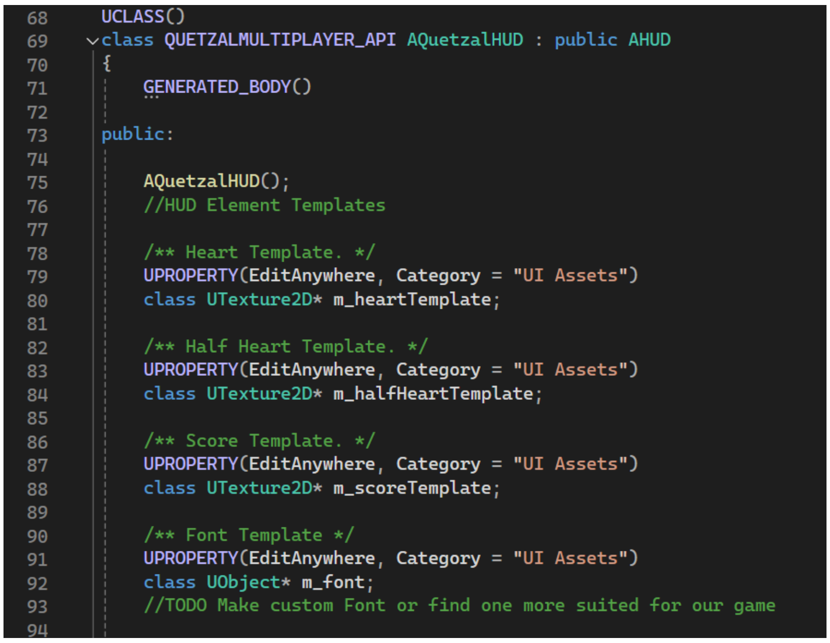
```cpp

#pragma once

#include "CoreMinimal.h"
#include "Widgets/SCompoundWidget.h"

/**
 *
 */
class QUETZALMULTIPLAYER_API SHealthWidget : public SCompoundWidget
{
public:
    SLATE_BEGIN_ARGS(SHealthWidget)
        :_HeartImage(nullptr)
    {}
        SLATE_ARGUMENT(UTexture2D*, HeartImage)

    SLATE_END_ARGS()

    /** Constructs this widget with InArgs */
    void Construct(const FArguments& InArgs);

protected:
    FSlateBrush HeartBrush;
};

```

- In the .cpp file set the resource we will be using (in this case use the parameter we take in to set the brush to the texture we will select)
- Set the size of the image to the width and height and set the draw type as an image
- Finally, to set up the widget we will create a child slot and create a new Slate Image using our brush (here you can also add an offset and many other functionalities but for our case, we just want an image)



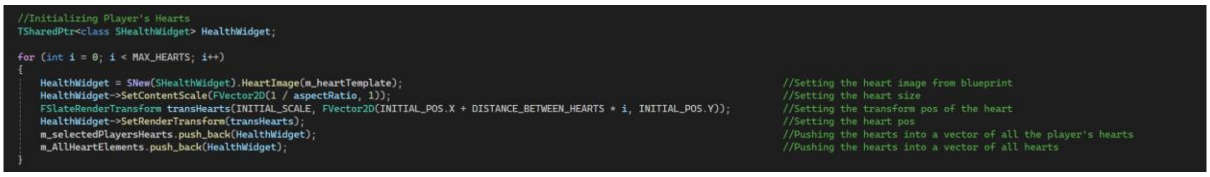
```cpp
BEGIN_SLATE_FUNCTION_BUILD_OPTIMIZATION
void SHealthWidget::Construct(const FArguments& InArgs)
{
    //This creates sets the brush that is established in the Blueprint for QuetzalHud
    HeartBrush.SetResourceObject(InArgs._HeartImage);
    HeartBrush.ImageSize.X = InArgs._HeartImage->GetSurfaceWidth();
    HeartBrush.ImageSize.Y = InArgs._HeartImage->GetSurfaceHeight();
    HeartBrush.DrawAs = ESlateBrushDrawType::Image;


    //This creates new UI elements with the brush initialized above
    //Most (if not all) of the animations will be done through QuetzalHUD.cpp where the HUD elements are created
    ChildSlot
    [
        SNew(SImage)
            .Image(&HeartBrush)
    ];

}
END_SLATE_FUNCTION_BUILD_OPTIMIZATION

```

- In our custom HUD class, create a UPROPERTY UTexture2D* variables to store the images we will create and display on screen.



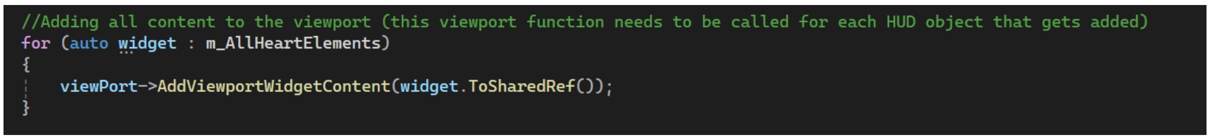
```cpp
UCLASS()
class QUETZALMULTIPLAYER_API AQuetzalHUD : public AHUD
{
    GENERATED_BODY()

public:

    AQuetzalHUD();
    //HUD Element Templates

    /** Heart Template. */
    UPROPERTY(EditAnywhere, Category = "UI Assets")
    class UTexture2D* m_heartTemplate;

    /** Half Heart Template. */
    UPROPERTY(EditAnywhere, Category = "UI Assets")
    class UTexture2D* m_halfHeartTemplate;

    /** Score Template. */
    UPROPERTY(EditAnywhere, Category = "UI Assets")
    class UTexture2D* m_scoreTemplate;

    /** Font Template */
    UPROPERTY(EditAnywhere, Category = "UI Assets")
    class UObject* m_font;
    //TODO Make custom Font or find one more suited for our game

```

- For each heart we want to make we need to:
  - Set the image from a template
  - Set the content scale using the aspect ratio to ensure HUD elements don’t stretch when the aspect ratio changes (see getGameResolution function),
  - Set the transform position
  - Add the hearts to a vector of the player hearts (to be able to modify them if the player takes damage)
  - Add the hearts to a list of all the heart elements (to be able to move them off screen when needed)



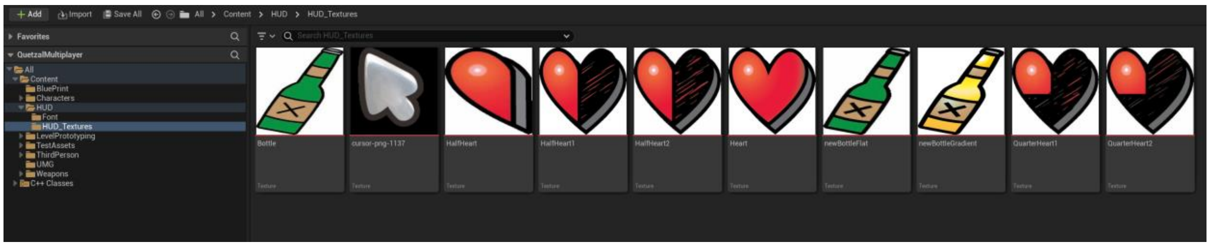
```cpp
//Initializing Player's Hearts
TSharedPtr<class SHealthWidget> HealthWidget;

for (int i = 0; i < MAX_HEARTS; i++)
{
    HealthWidget = SNew(SHealthWidget).HeartImage(m_heartTemplate);                                                    //Setting the heart image from blueprint
    HealthWidget->SetContentScale(FVector2D(1 / aspectRatio, 1));                                                     //Setting the heart size
    FSlateRenderTransform transHearts(INITIAL_SCALE, FVector2D(INITIAL_POS.X + DISTANCE_BETWEEN_HEARTS * i, INITIAL_POS.Y));  //Setting the transform pos of the heart
    HealthWidget->SetRenderTransform(transHearts);                                                                    //Setting the heart pos
    m_selectedPlayersHearts.push_back(HealthWidget);                                                                  //Pushing the hearts into a vector of all the player's hearts
    m_AllHeartElements.push_back(HealthWidget);                                                                       //Pushing the hearts into a vector of all hearts
}
```

- All widgets must be initialized through the game’s viewport

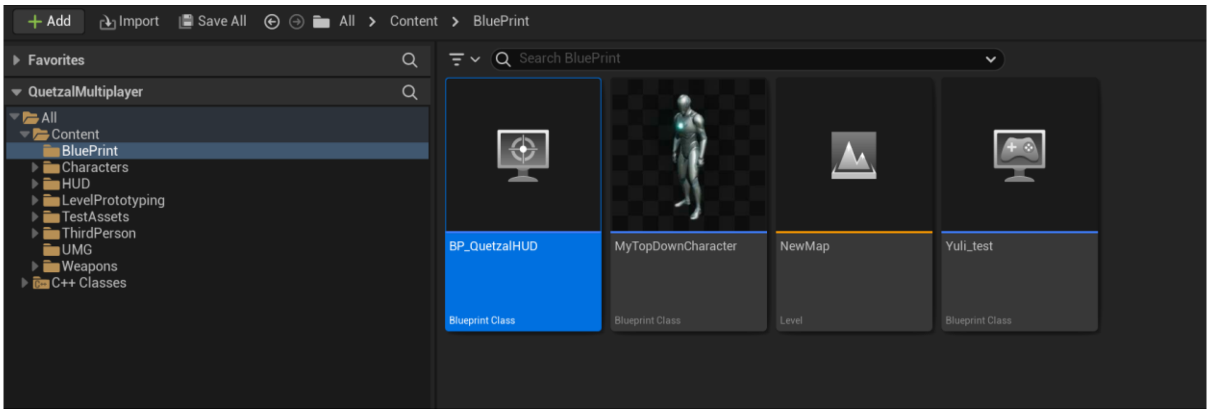


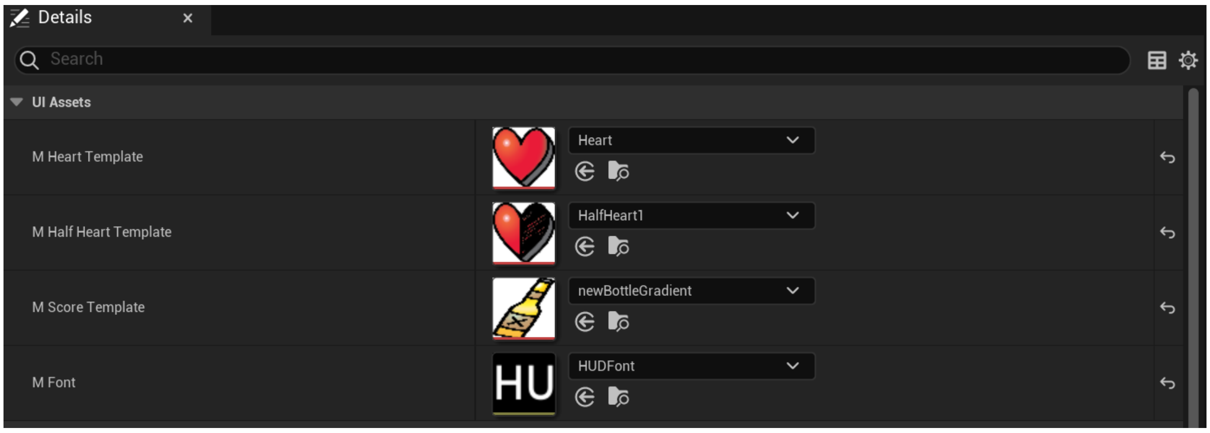
```cpp
//Adding all content to the viewport (this viewport function needs to be called for each HUD object that gets added)
for (auto widget : m_AllHeartElements)
{
    viewPort->AddViewportWidgetContent(widget.ToSharedRef());
}
```

- Import your HUD assets



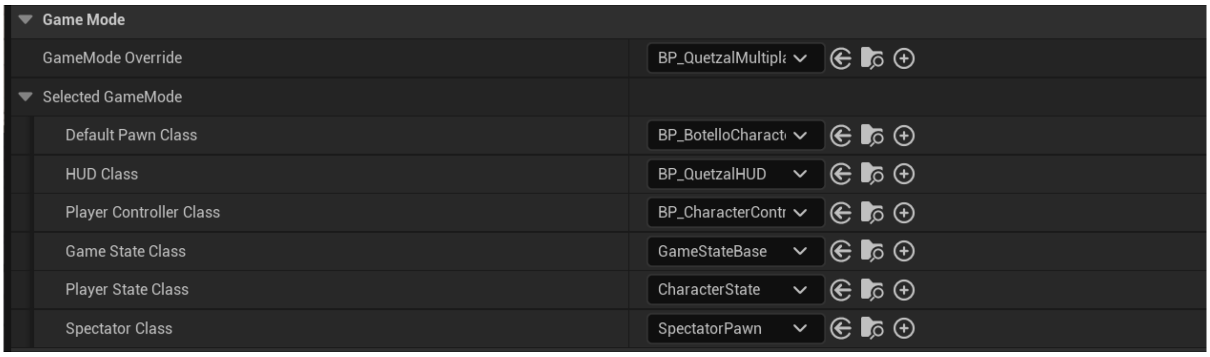
- In our BP_QuetzalHUD, select the variables created in our custom HUD.h file and set the textures we made to our texture variables.





M Heart Template	Heart
M Half Heart Template	HalfHeart1
M Score Template	newBottleGradient
M Font	HUDFont

- Finally set the custom HUD to the used HUD class in your Game Mode override



Game Mode

GameMode Override	BP_QuetzalMultipla
Selected GameMode	
Default Pawn Class	BP_BotelloCharact
HUD Class	BP_QuetzalHUD
Player Controller Class	BP_CharacterContr
Game State Class	GameStateBase
Player State Class	CharacterState
Spectator Class	SpectatorPawn