UNREAL FAST TRACK
WORKSHOP THREE

Workshop Three: Creating HUDs, Health Systems, and High Scores

Learning Points

Test Drive (60 minutes)

- Introduction to UMG
- Creating UIs and HUDs in Unreal Engine
- Adding menu systems and pausing to your game

Grand Prix (90 minutes)

- Making a more complex HUD
- Developing a health, damage, and score system
- Creating a Level-completion system

Off-Roading and Discussion (90 minutes)

- Re-creating UIs
- Adding unique mechanics to your game

Test Drive

For the third week, we will be focusing on UMG, which stands for "Unreal Motion Graphics." Understanding how the UMG system works is incredibly important for creating UI elements. The course we will be using is "Your First Hour with UMG." You can find the course at https://www.unrealengine.com/en-US/onlinelearning-courses/your-first-hour-with-umg.

This course has a project you have to redeem on the Marketplace to download. At www.unrealengine.com, log in and hover over your display name in the top-right corner. Click on **Personal**. This should bring you to your account page with a list of options on the left. At the bottom of the list is "**Redeem Code**". Paste the code found in the course there and you will unlock the project in your library.

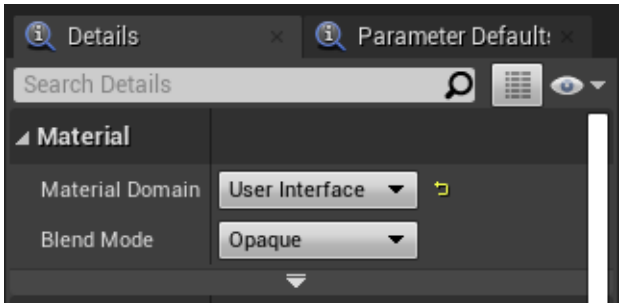
Grand Prix

First, using exactly what you just learned in the course, add a main menu and pausing to the game you made. It'll make the game experience feel more whole. Adding the Levels and Game Modes to your project will take a bit of time, and it will be worth it! Be careful when adding and changing Game Modes.

After the menu system is in place, we are going to use UMG and Blueprints to add score, player health, and an end goal to your Level.

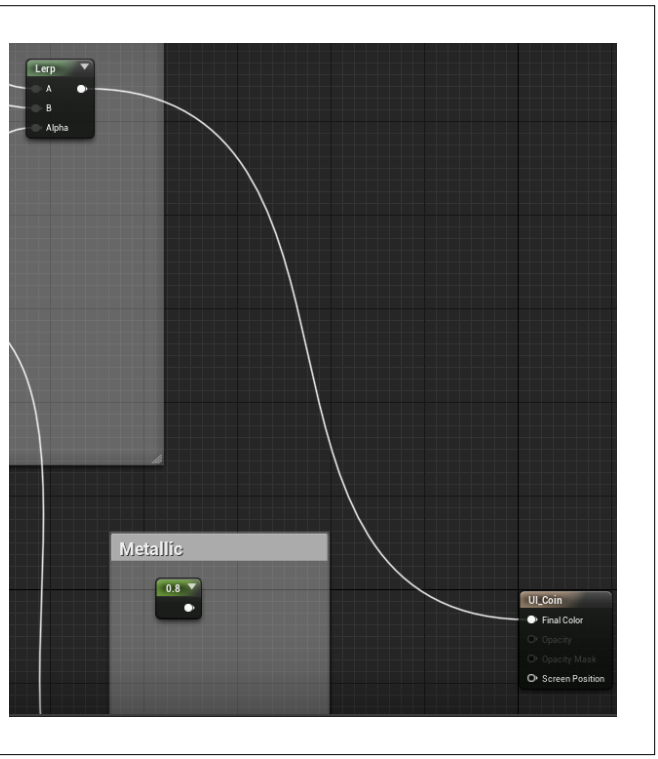
Score

The score system will be very similar to the ammo system created in the Unreal Online Learning course. Score will be determined by how many coins the player collects. You can either make coins rare and obtainable only through completing difficult side challenges or make them something that can be used to show the golden path to completing the Level.



First, we need a good image to use.

1. In the **Content Browser**, search for "**M_Pickup_Coin**". Right-click on it and duplicate the Material. Call the new Material "**M_UI_Coin**" and open it up.
2. In the **Details** panel on the bottom right, under "**Material**", change **Material Domain** **User Interface**". In the graph, find the **Lerp** node and connect it's output execution pin to the **M_UI_Coin** node's **Final Color** pin. Save the Material.



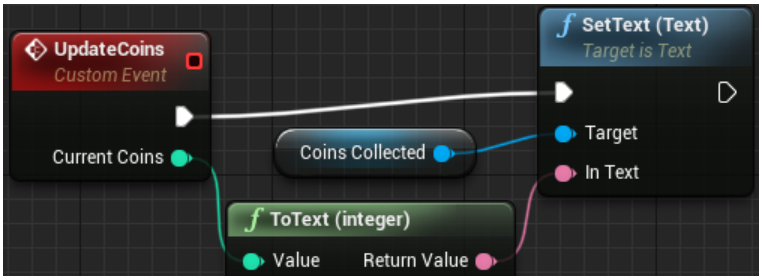
Now that we've created a UI Material, we need to create a HUD to add it to.

1. Back in the **Content Browser**, just like in the course, create a Widget Blueprint called "**UMG_HUD**" and open it up.
2. Add an **Image** widget to the **Canvas** panel and anchor it to the bottom right, with the **X** and **Y** values of **Alignment** set to "**1**, **1**". Check **Size To Content**. Under "**Appearance**" in the **Details** panel, change **Image** to "**M_UI_Coin**" and set the **X** and **Y** values of **Image Size** to "**160**, **160**".
3. Add a **Text** widget to the **Canvas** panel. Name it "**Coins Collected**" and check **Is Variable**. Anchor it to the bottom right, too. Set **Position X** to "**–65**", **Position Y** to "**–50**", and the **X** and **Y** values of **Alignment** to "**1**, **1**". Check **Size To Content**.
4. Set **Text** to read "**0**". Change the **color** to **black**, and the **font size** to **36**. Then change **justification** to **centered**.

You just created the coin section of the HUD. Let's make the HUD update when the player picks up a coin.

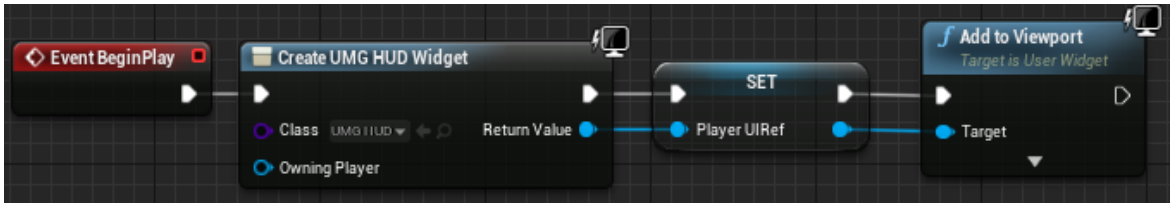
1. In the **UMG_HUD** Event Graph, add a custom event called "**UpdateCoins**". In the **Details** panel for the event, add an **Integer** input called "**Current Coins**".
2. Drag and drop a reference to the **Coins Collected** Text widget onto the graph. Off the reference node, add a **Set Text** node.
3. Connect the **UpdateCoins** node's execution and **Current Coins** pins to the two matching, empty pins on the **Set Text** node. Compile the Blueprint.

> **The HUD is Cosmetic**
> You'll notice that we don't suggest that the HUD contains much coding. The HUD simply receives the updated data and displays it; the data itself is updated elsewhere. At Epic, one of our best practices is adhering to the principle that the HUD is cosmetic. This means the HUD shouldn't do more than look pretty and tell the player what they need to know. The other parts of the game do the data storage and computation.

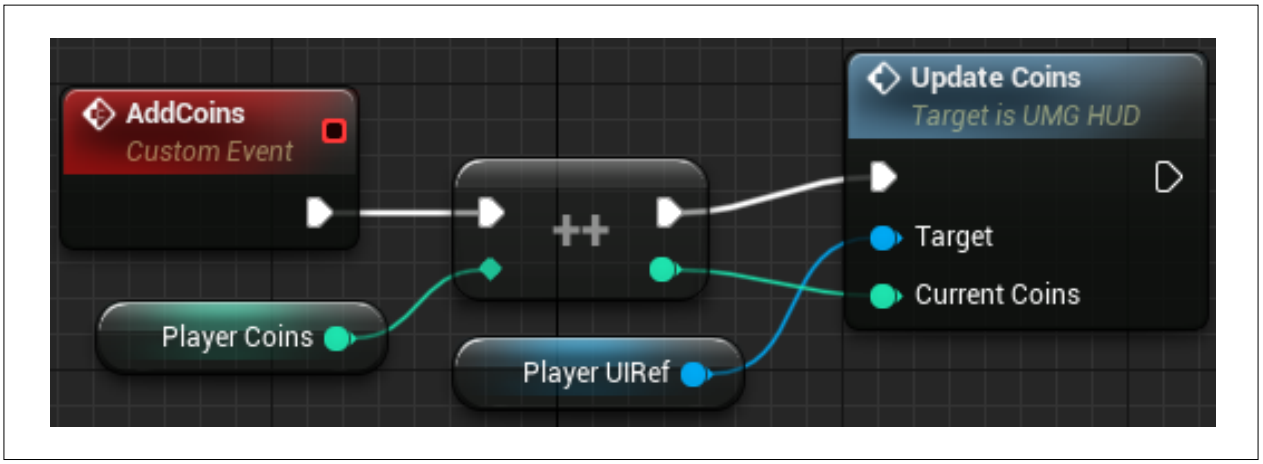


The HUD now has a way of updating when we call the **UpdateCoins** event. Let's add calling that event to our game. This next part takes place in the **Game Mode** Blueprint; it is good practice to keep the game data and Game State information in the Game Mode.

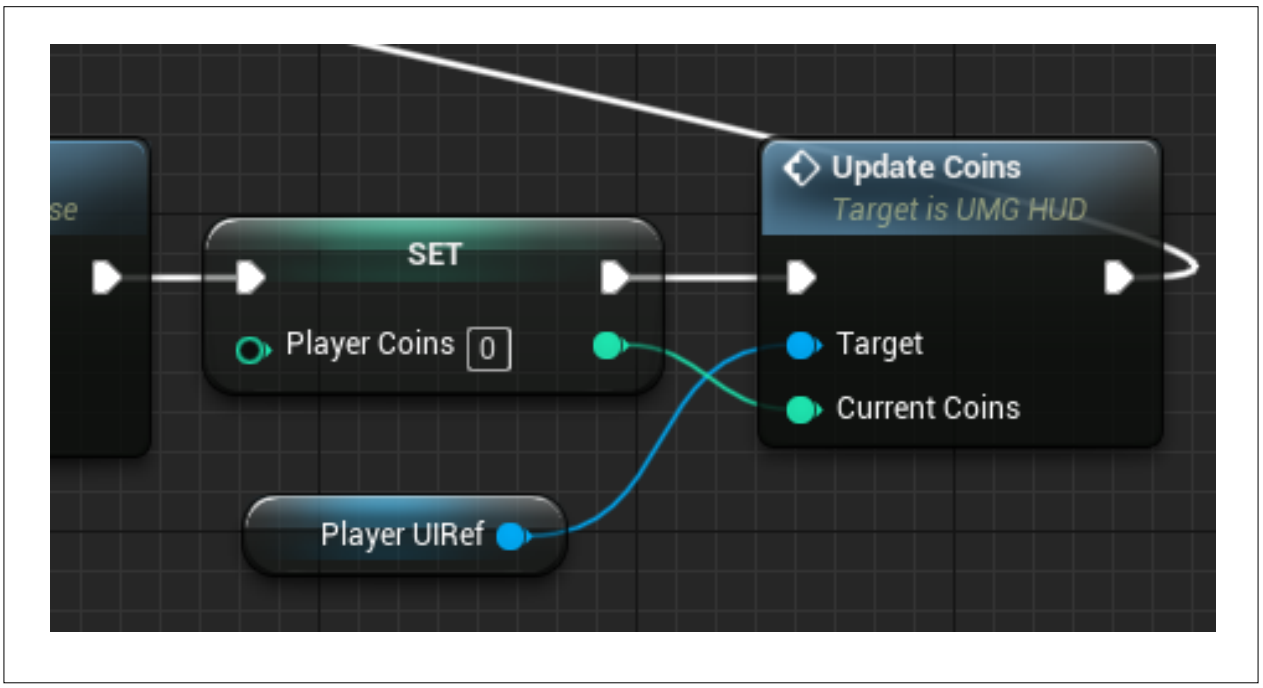
1. **This is the most important step**. To start, just like in the course, we are going to add the HUD to the screen and create a variable reference for us to use with the following node setup. We are doing this in the **Game Mode** Blueprint so we don't run into the problem of creating multiple HUDs when the player respawns.
   Be sure to add the nodes after **Event BeginPlay** and before the **Bind Event to On Destroyed** node you made in the first workshop. Don't forget to connect them back to the binding node, too.



2. Press **Play** to ensure that the HUD is showing up in the Viewport as expected.
3. Create an **Integer** variable called "**PlayerCoins**".
4. Add a custom event called "**AddCoins**" to the Event Graph.
5. Add an **Increment Integer** node. This can easily be found by searching for "**++**".
6. Add a **PlayerCoins** reference and connect it to the **Increment Integer** node's green input pin.
7. Drag and drop a reference to **PlayerUIRef** onto the graph. Off the reference node, add an **Update Coins** node. Connect the **Update Coins** node's **Current Coins** pin to the **Increment Integer** node's green output pin. Then connect the execution pins.

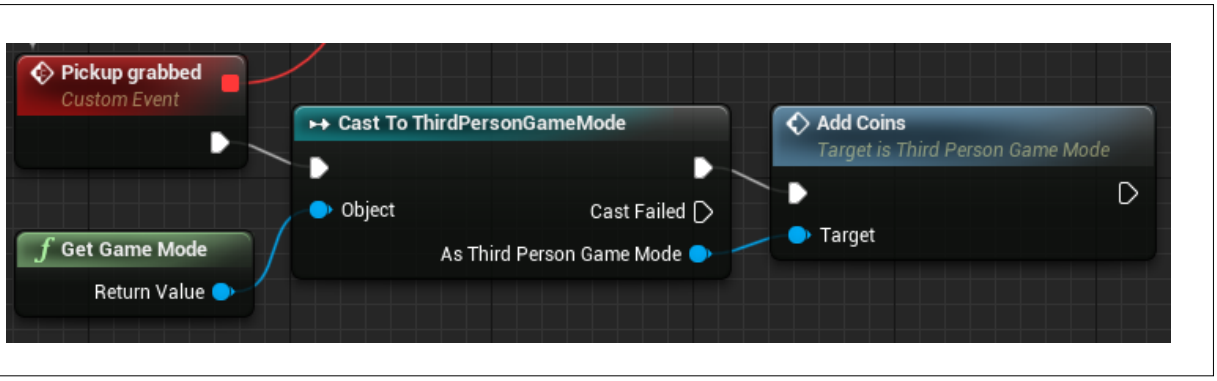


8. We also need to make sure the **PlayerCoins** variable resets to zero when the player dies. At the end of the **On Destroyed** event that we made in the first workshop, add a **Set PlayerCoins** variable node and set it to "**0**".
9. Next, similarly to above, drag and drop a reference to **PlayerUIRef** onto the graph and then add an **Update Coins** node. Connect the **Current Coins** pin to the **Set PlayerCoins** node's green output pin and connect the execution pins.



We're almost there!

1. Find "**BP_Pickup_Child_Coin**" in the **Content Browser** and open it up.
2. Find and delete the **Destroy Actor** node and the **Print String** node.
3. Add a **Get Game Mode** node. Off that, add a **Cast To ThirdPersonGameMode** node. Connect its input execution pin to the **Pickup grabbed** event's output execution pin.
4. Next, off the C**ast to ThirdPersonGameMode** node's **As Third Person Game Mode** pin, add an **Add Coins** node. Connect the node's input execution pin to the **Cast To ThirdPersonGameMode** node's output execution pin. Compile the Blueprint.



Now when the player picks up a coin, it should be added to the total number of coins recorded in the bottom right of the HUD, and the number should be reset to zero when the player dies! Add coins to your Level where you see fit!

Separation of Concerns
Separation of concerns is a key design principle in coding. It simply means that all aspects of your code should have specific responsibilities, and it should be clear and make sense which part of your program is in charge of what.

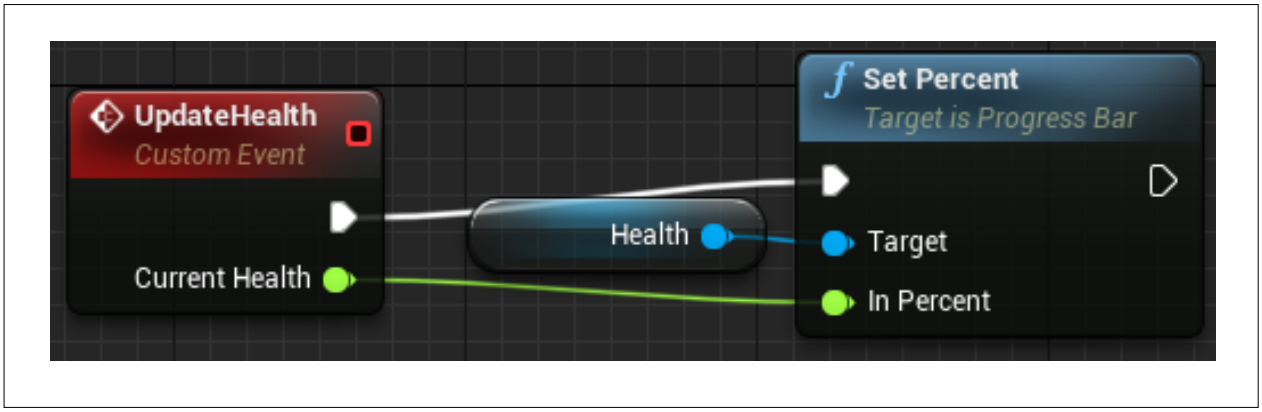
For example, in real life, when you pick up a coin, the coin doesn't tell you how many coins you have. All it tells you is that you've picked up a coin. Your wallet holds all of the coins you've collected, and your mind does the math to figure out how much money you have. Each responsibility and concern is separate in a way that makes sense.

Following this principle makes things more complex in the beginning of a game project, but makes it much more maintainable as the project grows.

Health

For recording player health on our HUD, we are going to use a health bar rather than just a number.

1. Open up **UMG_HUD**.
2. Add a **Progress Bar** widget anchored to the bottom right of the **Canvas** panel, and in the **Progress** section of the **Details** panel, set **Percent** to "**100**". Name the widget "**Health**" and check **Is Variable**.
3. In the Event Graph, add a custom event called "**UpdateHealth**". In the **Details** panel for the event, add a **Float** input called "**Current Health**".
4. Drag and drop a reference to the **Health** Progress Bar widget onto the graph. Off the reference node, add a **Set Percent** node.
5. Connect the **UpdateHealth** node's execution and **Current Health** pins to the two matching, empty pins on the **Set Percent** node. Compile the Blueprint.

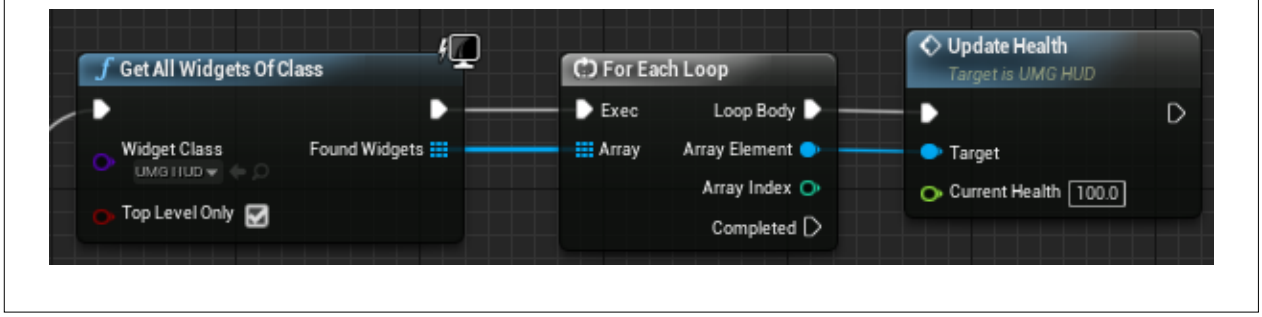


Now that we've added a way to update player health on the UI, let's give our character some health. This next part will be in the **ThirdPersonCharacter** Blueprint.

1. Create a **Float** variable called "**PlayerHealth**" and set the default value to "**100**".
2. Next, create a custom event called "**PlayerDamaged**". Off the **PlayerDamaged** event, we are going to set **PlayerHealth** to the new amount. First, add a **PlayerHealth** reference, and then create a **float** – **float** node off it. Set the bottom number (called the subtrahend) on the node to "**5**" and then connect the result of the **float** – **float** node to a **Set PlayerHealth** node.
3. Now we are going to do a new trick. Add a **Get All Widgets Of Class** node to the graph. Change the **Widget Class** setting to "**UMG_HUD**".

This is an alternative to the UI reference we made in the **Game Mode** Blueprint. It's a convenience function that returns an array (a numbered list) of all the running widgets of the specified class. This will let us update multiple UIs at once if we need to and is a method used in professional games.

4. Drag a wire from the **Get All Widgets Of Class** node's **Found Widgets** pin and add a **For Each Loop** node.
5. Off the **For Each Loop** node's **Array Element** pin, add an **Update Health** node and connect the execution pins.



A **For Each Loop** node will run a series of nodes for every item in the array given to it. In this case, the **For Each Loop** node will find every widget with the **UMG_HUD** class. It will then run the **UpdateHealth** event for each one. Now we just need it to get the correct health amount.

6. The **Health** Progress Bar widget that will be updated with the **UpdateHealth** event wants a percentage. Because of that, we need to divide **PlayerHealth** by 100 to change the amount into a percentage. Off the **Set PlayerHealth** node's green output pin, add a **float ÷ float** node. Change the bottom number to "**100**", and plug the **float ÷ float** node into the **Update Health** node's **Current Health** pin.

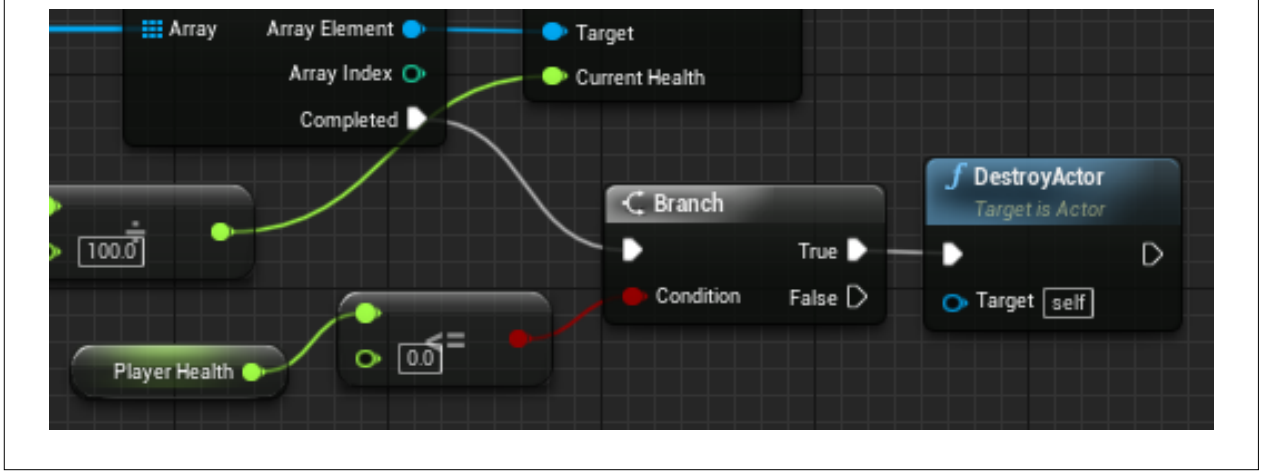
Health does not mean anything if the player doesn't die when they reach zero! We are going to add that in now.

1. Drag a wire from the **For Each Loop** node's **Completed** pin and add a **Branch** node.

The **Branch** node connected to the **Completed** pin is run after the **For Each Loop** node has run through all of the nodes connected to its **Loop Body** pin for each item in the array. In this case, we just have one item in the array.

2. The **Branch** node should check to see if **PlayerHealth** is equal to or less than zero. Add a **float <= float** (less than or equal to) node off the **Branch** node's **Condition** pin. The **float <= float** node's top pin should be connected to a **Get PlayerHealth** pin, and its bottom pin should be set to "**0**".
3. Off the **Branch** node's **True** pin, simply place a **Destroy Actor** node. Now your player will be destroyed (and respawn!) after taking too much damage.

We can't forget to reset **PlayerHealth** back to "**100**" when we respawn!

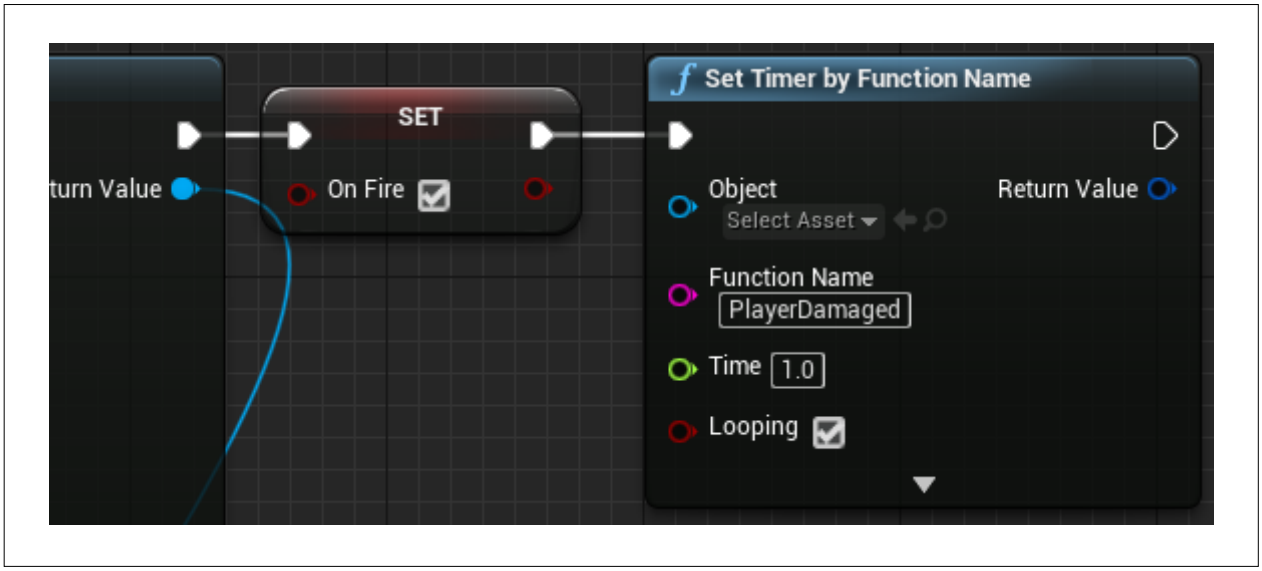


4. Find the **Event BeginPlay** node.
5. Just like above, add a **Get Widgets Of Class** node, set **Widget Class** to "**UMG_HUD**", and attach the **For Each Loop** node.
6. Once again, off the **For Each Loop** node's **Array Element** pin, add an **Update Health** node and connect the execution pins.
7. Finally, add a **PlayerHealth** reference, and then use a **float ÷ float** node to divide **PlayerHealth** by 100 like you did earlier, and attach the **float ÷ float** node to the **Update Health** node's **Current Health** pin.

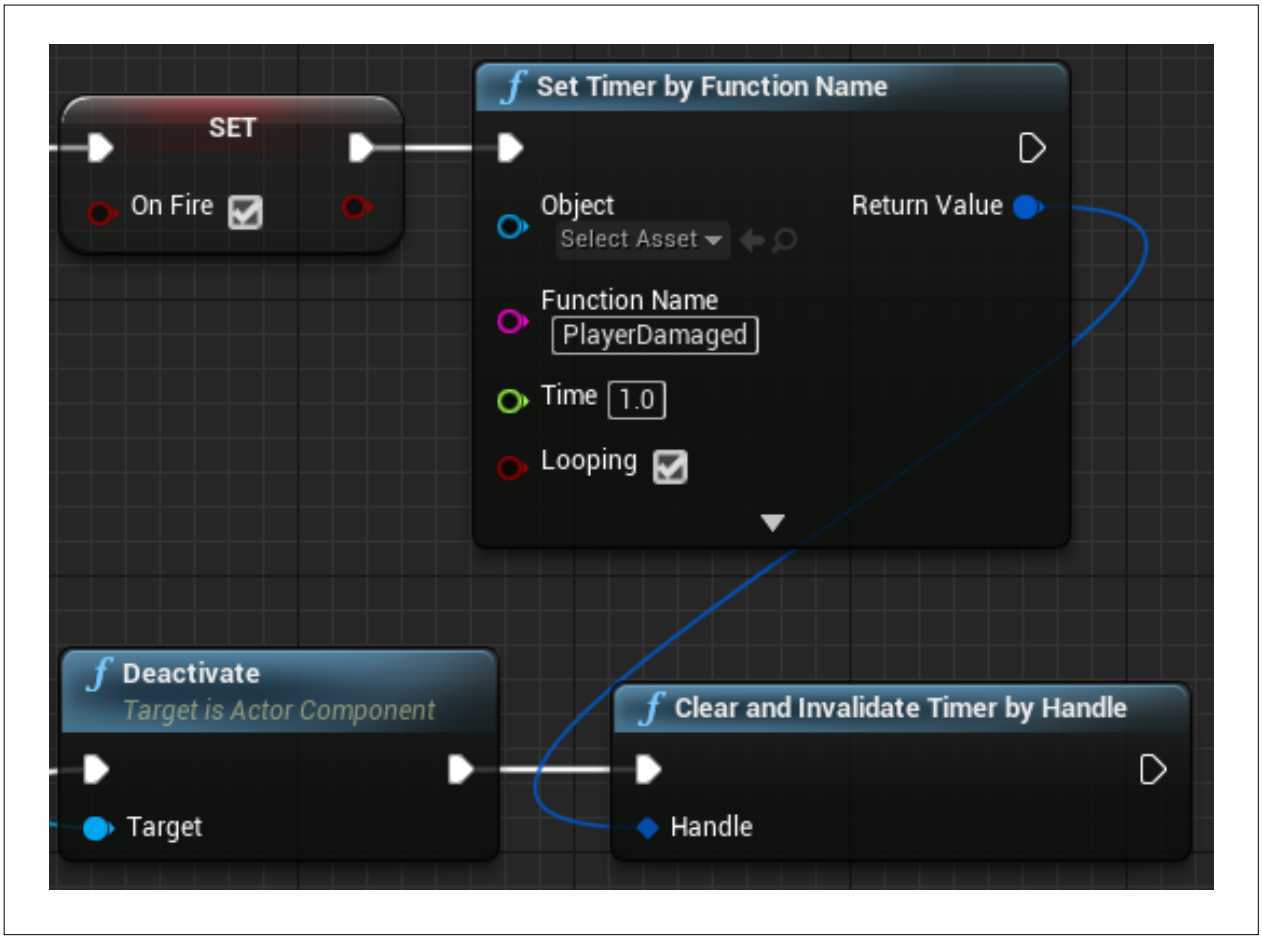
Damage

We've made a way for the player to get damaged, so now we need something to damage the player with! Let's make our fire from the last workshop do it!

1. Find the **On Component Begin Overlap** event you made in the last workshop for lighting the player on fire.
2. At the end of that chain, off the **Set OnFire** node, add a **Set Timer by Function Name** node. Type in "**PlayerDamaged**" for **Function Name**, set **Time** to "**1.0**", and check **Looping**.



3. At the end of the On Component End Overlap event, off the **Deactivate** node, add a **Clear and Invalidate Timer by Handle** node. Then connect that node's **Handle** pin to the **Set Timer by Function Name** node's **Return Value** pin.



Now the fire will damage the player every second the player is on fire, and stop damaging them when the fire stops.

End Goal

To add the end goal, we are going to migrate an asset from the "Your First Hour with UMG" project.

1. In **Content → FirstHourUMG → Pickups**, find the **Part_LevelEnd Cascade** particle system. Right-click on it and choose **Asset Actions → Migrate** to migrate the file to your Grand Prix project's **Content** folder. Once you've migrated the asset, open up your Grand Prix project.
2. Create a new Blueprint Actor. Name it "**BP_LevelEnd**".
3. Add a **Cube** component to the Blueprint with its **Scale** property set to "**2.0**, **2.0**, **0.1**". Note the decimal points. Make the **Cube** component the root component.
4. Under the **Materials** section in the **Details** panel, select "**M_Pickup_Coin**" from the dropdown menu.
5. Add a **Box Collision** component to the center of the Blueprint with the Scale property set to "**1.0**, **1.0**, **20.0**".
6. Add a **Particle System** component to the center of the Blueprint and set its **Scale** property to "**1.0**, **1.0**, **10.0**". In the **Details** panel, under "**Particles**", choose the **Part_LevelEnd** template. Compile the Blueprint.
7. Back in the **Content Browser**, find and open the **Part_LevelEnd** template.
8. On the left side of the **Emitters** panel, right-click and go to **TypeData → New Mesh Data**. Select "**Mesh Data**" from the list, and in the **Details** panel change the mesh to "**SM_Pickup_Coin**".
9. Now select "**Initial Size**" from the list, and find "**Max**" and "**Min**" in the **Details** column. Set **Max** to "**0.2**" for **X**, **Y**, and **Z**. Set **Min** to "**0.01**" for **X**, **Y**, and **Z**. For **Locked Axes**, select "**XYZ**".
10. Save the template. Your Blueprint should now be spawning coins of various sizes.

Unreal Engine is a very large and complex tool. Even experts at the engine do not know every single thing about every part of it. The key to coding and developing in Unreal Engine is not having a mastery of everything; the key is understanding the tool at a base level well enough that you can think of a concept, do research, search around, and figure out how to do what you need to do. There's often no one "right" answer.

11. In the **BP_LevelEnd** Event Graph, use everything you have learned so far to add the logic for winning. Think about the interaction between the player, the Game Mode, the UMG, and the goal. Where will you store the data? How will the different aspects speak to one another?

If you cannot figure out how to add this logic into your game, talk with your teammates, look back at the "Your First Hour with UMG" project files, and do research to see how others have done it.

Off-Roading and Discussion

Epic provides over 20 premade widgets for use with the widget designer. Look through all of them and try to add different widgets to your UMG. List some intuitive user interfaces of games that you really like, and think about how you could re-create those features using UMG. Discuss with your group different UIs from different games and think about how they can be re-created. Have each person pick a different game from which to show screenshots and gameplay examples.

If you can't think of an example, the Active Reload mechanic from Gears of War is a great choice. Gears of War, which was made by Epic using Unreal Engine, introduced an active style of reloading to the genre. Instead of reloading being a passive experience where players hit the reload button and waited for the bar to refill, Gears of War added a target partway through the bar. If the player hit the reload button again while the progress bar was within the target, the character would reload their gun much faster. You could remake that mechanic using UMG!

As your team is looking at great examples and everyone is making their own UI features, discuss with your teammates what makes for a good user interface. What do the examples you all share have in common? What unique features do some user interfaces have that add to the experience?

After you've discussed user interface design and re-created some features using UMG (or made your own!), you will probably have an idea for a small mechanic or system you'd like to see in the game you are building. Discuss with your teammates some small ideas you have for the game you've been making. **You should be adding something very small in scope.** After the discussion, everyone should try building their mechanic or system into their game! Explore the various systems within Unreal Engine that you've learned about so far. Do research on areas you aren't familiar with yet, and learn how to implement the idea you have.

Once you've added the mechanic or system, show it off to your teammates. Play each other's games if you can and see how it feels! Use the 3 **D**s as a framing for the discussion:

- **Difficulties**: What was the most difficult or confusing part of this week's workshop? Was the mechanic or system you added small in scope?
- **Discoveries**: What did you discover while making your UI elements? How about while making the mechanic or system you added?
- **Dreams**: As you fleshed out your game, what bigger ideas for mechanics and systems popped into your mind? What would you need to know to execute those big ideas?

CONGRATULATIONS!

You've finished Workshop Three of the Fast Track!

The Pathless | Giant Squid