Udon SDK Simple Examples

This directory contains some sample Udon programs in a scene to get you started. Open the scene ‘UdonExampleScene’ to explore.

There are five areas to explore – Prefabs, Cubes, Udon Variable Sync, PlayerDetection and PlayerAudio.

Prefabs

- **VRCWorld** has the typical components needed to upload your world, and it has a special UdonBehaviour on it with four public variables: **jumpImpulse**, **walkSpeed**, **runSpeed** and **strafeSpeed**. Take a look at the graph to see how these variables are set for the local player on **Start**. That means you can set them in the inspector and they will be set for each player in your world when they join.

- The **MirrorSystem** is a group of a few assets set up to easily toggle a **VRCMirror** on and off. It has two child objects – **MirrorCanvas**, which is a World Space UI Canvas with a **UIToggle**, and the **VRCMirror** prefab, with no special changes made. Take a look at the UdonBehaviour on the **MirrorSystem** object. This is program is called **ToggleGameObject,** and it can be used to turn any GameObject on and off. It has a public variable for a **targetGameObject** – this is set to **VRCMirror** for the example, but you could easily drop another GameObject in its place. When this program received a **CustomEvent** called “Toggle”, it will check whether the **targetGameObject** is currently active and sends this value through ‘**Op Unary Negation**’, which just flips the value to its opposite. So a value of true becomes false and vice versa. Then this new flipped value is used to set the new value of the GameObject, flipping its Active state each time the Toggle event is received. How does the UIToggle fire this ‘Toggle’ event? Take a look at the UIToggle under MirrorSystem>MirrorCanvas>MirrorToggle. This is a standard UIToggle component, with an event wired up – under **OnValueChanged**, we’ve added an Action to target the MirrorSystem’s UdonBehaviour and fire its SendCustomEvent method with the string ‘Toggle’. The final important detail is that the MirrorCanvas object has a **VRCUIShape** component attached. This tells VRChat to enable Interaction with all UI items on this canvas.

- The **AvatarPedestal** is a simple working avatar pedestal. You’ll have to do a ‘Build and Test’ in the VRChat SDK window under ‘Builder’ in order to see it working. The prefab itself is a cube with a **VRC Avatar Pedestal** component with a public Blueprint Id set, and ‘Change Avatars On Use’ turned on. There is an UdonBehaviour on this object, open it up to see how the behaviour listens for an ‘Interact’ event, then uses

GetComponent to fire the **SetAvatarUse** command for the **Local Player**.

- To activate the **Station,** you'll need to Build and test, and then walk your avatar to the chair object and **Interact** with it (typically your **Trigger** or **Left Mouse** button). This has a very simple **UseStationOnInteract** program that gets the local player object and calls Use Attached Station.

Cubes

- The **On Mouse Down** cube will switch between 3 materials when you click on it in the editor. Take a look at its two attached UdonBehaviours: **SendEventOnMouseDown** and **ChangeMaterialOnEvent.**

- The **Timer** cube automatically changes between its 3 materials based on a **duration** variable you can change in the inspector before you hit play. It does this with two UdonBehaviours: **SendEventOnTimer** and **ChangeMaterialOnEvent** (the same exact script as on the On Mouse Down cube).

- The Click for Loops cube will change its text to read something like 'loops:012345678'. It does this by running a loop X number of times and adding to the UI Text Field. It's got a **SendEventOnMouseDown** UdonBehaviour just like the first cube, but it points to another component. Click on the **target** public variable on this UdonBehaviour to highlight the Text field that is being changed. Click on this text field and you'll see a **SimpleForLoop** UdonBehaviour. You can change the **numberOfLoops** variable before running the scene to change the text it creates.

You'll need to Build & Test a local version of the scene so it can run in the VRChat Client in order to test the next group:

- To swap the materials on the **Interact Cube**, walk your avatar to it and **Interact**. You may have guessed – a **ChangeMaterialOnEvent** for the effect, and a **SendEventOnInteract** as a trigger.

- You can also walk over to the **On Pickup Cube** and press your pickup button (typically your **Grab** or **Left Mouse** button). Once your avatar is holding it, you can **Interact** with it to change its color. Take a look at the **PickupAndUse** program on the cube. It changes the color of the material instead of swapping it out entirely.

Udon Variable Sync

This area shows Sync working within VRChat in a few different ways. You'll need to run Build&Test with at least 2 clients to see this working – this opens up two instances of the VRChat world on your computer so you can test it. Make sure you to open the VRChat

SDK Window, switch to ‘Settings’, and set the VRChat Client path to the actual location of your VRChat installation. It’s probably something like ‘C:/SteamLibrary/steamapps/common/VRChat/VRChat.exe’.

- If you create the world, you will be the Owner of all the objects in the world, and you will have control over the **UIButton Owner** on the left. Every time you Interact with this button, its counter will increase for everyone in the room. Only you can push this button, unless you leave the room – which gives someone else ownership of all the objects. Take a look at the button in the hierarchy under Canvas/Panel/ButtonSyncOwner. Its OnClick event has been wired to the attached UdonBehaviour to fire a custom event also called OnClick. This doesn’t happen automatically, you have to wire it up yourself. Note that you don’t **have** to call the CustomEvent this name, as long as you use the same string in the UI Event as you do for the CustomEvent. Next, take a look at the Udon Graph to see how the **clickCount** is stored on the object and set from the event **OnDeserialization**. This event is called on the other people in the room when the variable is updated. The event is not called for the player who set the variable, so we wire the flow that comes out of **SetVariable** into the **Text.SetText** node so that changing the variable changes the Text for the Owner of the button as well.

- The **UIButton Anyone** on the right can be pushed by anyone in the room! All over the other UI items will only update when the Owner presses them. Upon interaction, that user becomes the owner of the button, which lets their instance become the source of truth for how many button clicks should be displayed. Take a look at the Udon Graph on this object to see how everything works. This graph does three things in a specific order:
  - When the **Interact** event fires, it sets the Owner of the Button to the local player – the one who triggered this event, and then updates the Text for this new Owner.
  - After the owner is set, it can set the **clickCount** variable. There is a known bug where this doesn’t work right away, so a new Owner will only have their clicks counted starting on the second click. This will be fixed soon.
  - Finally, with the **OnDeserialization** event, the Button’s **uiText** label is updated with the new count. This will happen for each player who is not the owner of the object.

- The **UISlider** can be controlled by the Owner – just aim and interact to change the value and it will sync its value to the other players in the room. This Udon Graph is very similar to the **UIButton Owner** and the rest of the UI examples in the scene. It uses public variables to wire up the UI and listens to the slider’s **OnValueChanged** to fire a Custom Event also called **OnValueChanged**. This prompts the graph to

save the current value of the slider to a synced variable, which is picked up by the other players **OnDeserialization**. It also updates its own text readout using this value.

- The **UIToggle** is one of the simplest examples, following our familiar formula – fire a custom **OnValueChanged** event from a UI Element's **OnValueChanged** event, update a synced variable, and update its own state from the synced variable.

- The **UIDropdown** works the same way as the above UI elements.

- The **UITextField** works very similar to the above elements. Note that you have the choice between subscribing to **OnValueChanged** or **OnEndEdit**. This example uses **OnValueChanged** to send updates more frequently, the other option would wait until an 'enter' command is made.

- The **PickupCube** on the left can be picked up by anyone in the room. Once it is picked up, it will change its color, and that new color will be synced to everyone else in the room. Take a look inside the attached UdonProgram, and notice that the Color data is synced using **smooth** interpolation. This helps smooth out the data over the network. Try the other modes and see what changes. This program uses an **Update** event to run on every frame – but the first thing it does in the **Block** node is check whether this player is both the **Owner** of the object and whether the object **Is Held**. If either of these are false, the **Branch** after the **Op Conditional And** node will be false, and that will end this flow, skipping to the second flow of the **Block** statement, which sets the color of the material for every player.

- The **PickupSphere** on the right can also be picked up by anyone. Its UdonBehaviour is empty! Instead of containing a program, it provides sync and ownership abilities just by using the checkboxes on the UdonBehaviour.

PlayerDetection

This area shows three different ways of detecting Collision events with the player.

- **PlayerTrigger** is the simplest and most common way, using a **Trigger Collider**. When the player steps into this collider, their name will be displayed in a Text field. In this example, the PlayerTrigger GameObject has a Box Collider with 'IsTrigger' checked, and an UdonBehaviour with our graph. This graph has two events – an **OnPlayerTriggerEnter** and an **OnPlayerTriggerExit**. When the events are triggered, they have a reference to the **VRCPlayerApi** object that called them. From this object, we can **Get** the **DisplayName** of the player, format it to say {name} Entered or Exited, and then **SetText** on a **Text** field to update it.

- **PlayerCollision** shows how you can make a moving physics object cause a Collision Event when it collides with the player. There are two main parts to this sytem – the **FireOnTrigger** graph on the **TriggerArea** GameObject, and the **Projectile** graph on the **Projectile** object. The result of this system is when a player steps into the **TriggerArea**, the **Projectile** is sent hurtling towards them. If it collides with them, it will vibrate their controllers and make their name appear in a Text field.

    - The **FireOnTrigger** graph is a great one to understand and reuse – when a Player enters its Trigger Collider, it will send an event to a target UdonBehaviour. Since the *eventName* and *target* are public variables, they can be changed in the inspector to fire any event on any UdonBehaviour, like opening doors, playing sounds, triggering animations. It also uses **Debug.Log** to log into your console whenever the event is fired so you can troubleshoot issues more easily. You can disconnect the Debug.Log event once everything is working as you'd like.

    - The **Projectile** graph is one of the more complex example graphs, but it's got lots of great logic that you can learn and use in your other graphs.
        - On **Start**, it saves the position of the object into a variable called *originalPosition*. We later use this to reset the position of the object.

        - On **Custom Event 'Fire'**, it does two things in order by using a **Block** node.
            - First, it triggers our **Reset** functionality, which is enclosed in a group for easier understanding. We fire the Projectile towards the player in order to test Collisions, so we need to make sure that the projectile is ready to be fired before we send it off each time. The Reset group will **Set the Position** of the Rigidbody (the physics element that moves this object) to the *originalPosition* that we saved at start. It will **Set the Velocity** and the **Angular Velocity** of the Rigidbody to Zero – which stops it moving and spinning. Finally, it turns off the **Constant Force** component which will stop applying force to the Rigidbody.
            - After everything in the Reset group above has run, our projectile should be sitting in its original position, ready to be fired again. To fire it, we turn on the **Constant Force** component, which applies forward motion. Since the player is standing roughly in front of it in order to trigger it, they're probably in its path.

        - On **PlayerCollisionEnter**, two main things happen – we write the Player's DisplayName into the Text Field, and we Vibrate their hands if they're using controllers with haptics, by sending **PlayHapticEventInHand** events to their left and right hands.

        - On **PlayerCollisionExit**, we write the Player's DisplayName into the text field and then call the Reset group. This way, the block should appear back in its original position once it's done hitting the player. We don't have to do this since we call Reset right before we fire, but gives a nicer visual to the Player, since they can see the projectile is ready to fire again right away.

- **PlayerParticleCollision** demonstrates how a ParticleSystem can trigger event when any of its particles collide with a player. It's also got two parts to the system.

    - The **TriggerArea** GameObject has a Box Collider with 'IsTrigger' turned on, and an UdonBehaviour with a graph called **SetActiveFromPlayerTrigger**. This is another highly-reusable graph.When a player enters the Trigger Collider, the *target* GameObject will be set to active, and when they exit, the *target* will be set to inactive. You can toggle any GameObject by setting the *target* of this graph in the inspector! In this case, it will toggle the **CollisionParticles** GameObject.

    - The **CollisionParticles** GameObject is inactive by default, so it starts in the 'off' state. It has a **ParticleSystem** component with the Collision module turned on, the *Collision Type* set to 'World', the *Mode* set to '3D', and the *SendCollisionMessages* option set to 'On'. It also has an **UdonBehaviour** with the **OnPlayerParticleCollision** event. This event is simpler than the others, it doesn't have Enter, Stay and Exit, but rather just a single event that is called when a particle collides with a player. For our example, we **Set** the **Text** on the *textField* variable to have the displayName of the player, as well as the time that the collision occurred so we can more easily see multiple collisions happening.

PlayerAudio

This setup shows how you can control some settings for each Player's Voice and Avatar Sounds.

- The **MuteCube** and **UnmuteCube** both have the same UdonBehaviour on them – **SetAllPlayersMaxAudioDistance**. This graph does a couple of things that are useful to understand separately:
    - It demonstrates how **VRCPlayerApi.GetPlayers** works. This is a highly useful node, you can use it to get an Array that references every player in your world so you can go through them one by one. In order to save memory and performance, you have to first create this Array. You can see this happen with the "Ctor" node, which is a Constructor for VRCPlayerAPI[]. In this case, we've created it with a size of 8, so it can hold up to 8 players. You want to set this to the Max Players allowed in your world. After you've create the array, you feed it *into* the **GetPlayers** node, and it's populated with all the players.
    - This graph then uses a **For** loop to run a function on each of the Players. The loop

will change its *index* output each time, so we plug that index into a **VRCPlayerApi[].Get** node, which will *get* the player at each *index* of the array.

- Once we have a reference to this player, we **need to check if it's really a Player** or just an empty reference. We do this by checking if the reference we got is Null, meaning it doesn't exist. We use the **Object !=** node to do this. The node will return **True** if the objects are not equal, so we can chain off the **True** output to do something with valid players.
- Finally, if we have have a valid player at this point in our For loop, we can set their **VoiceDistanceFar** and **AvatarFarAudioRadius** to the value of our variable **maxDistance**. Since this variable is public, we can easily change it in the Inspector and use the same graph for both Mute and Unmute. Mute will set the distance to 0, so you cannot ever get close enough to hear the Player, and Unmute will set it to 40, which is a bit further than the default of 25.