Avatar Components

Purpose

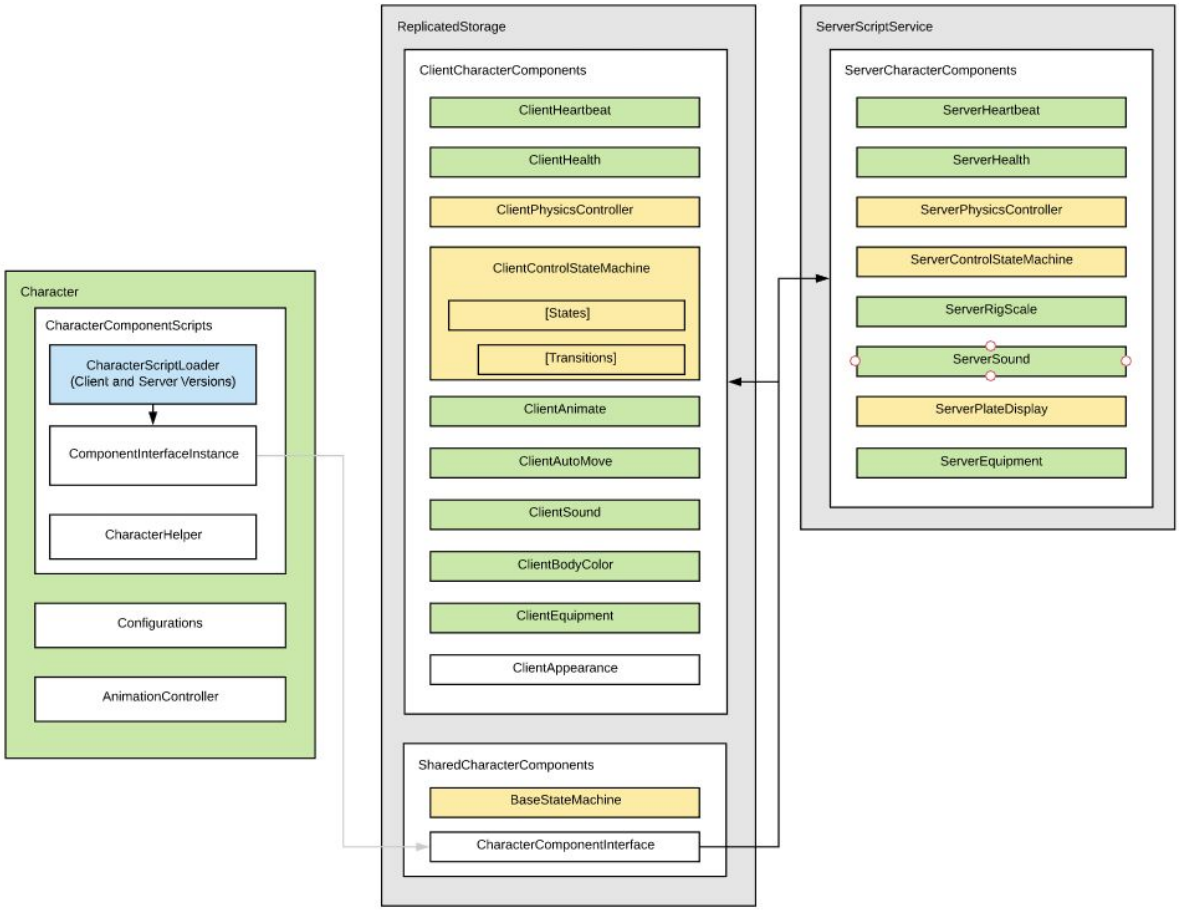
As developers continue to grow and stretch the Roblox system and the Avatars in particular, they are requesting to modify, replace and remove various aspects of the current Humanoid functionality. Currently, with the Humanoid object, this is very limited. The new component system allows developers the flexibility to customize the character functionality to the specific requirements of their game.

Component System

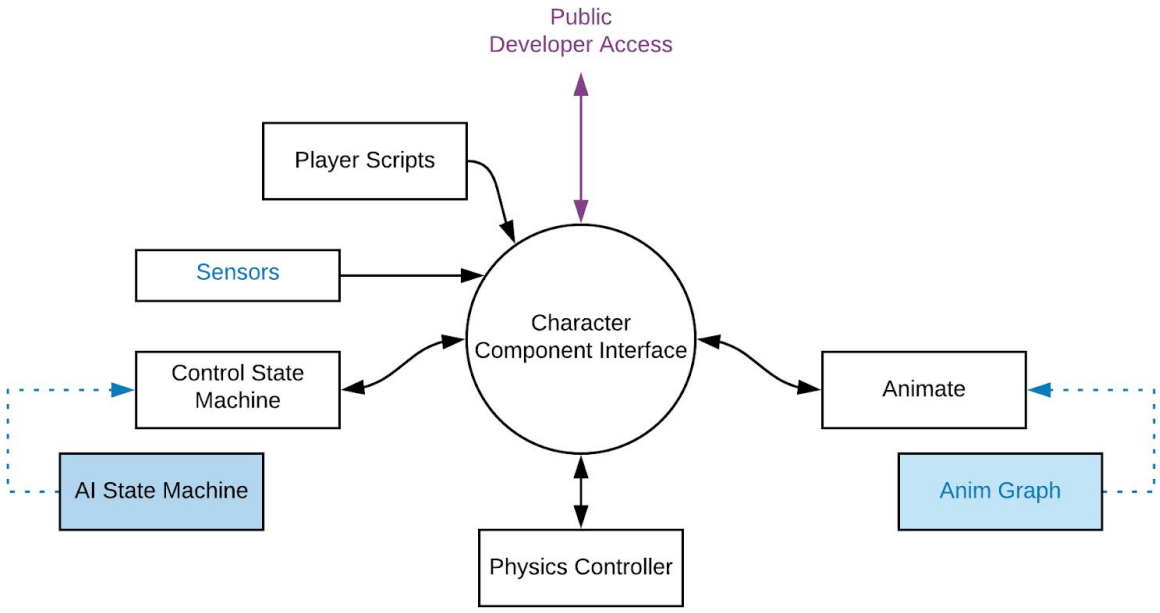
To facilitate the ability for developers to choose which Avatar functionality they want, including customizing, replacing or removing existing functions, current Humanoid functionality will be broken out into components. This change will separate logical functional groups into individual components and allow each to be customized, modified, replaced or even removed from each character instance.

Additionally, the components can be implemented in an incremental method, with new components being added to extend existing functionality without requiring developer changes to existing code. Developers will also be able to take advantage of componentized features as they are released, without requiring all of the Humanoid functionality to be updated before releasing individual parts.

To further allow developers to customize the character's behavior, components that help support Lua implementations. These versions of the components would provide hooks that scripts can integrate with that will allow the component's functionality to be implemented in Lua.



The character components will provide a common interface for all developer access that will allow the underlying components to be changed without requiring changes the code that refers to them. This way, the components can be modified or upgraded and, as long as the common interface exposed by them remains the same, consumers of the functionality of this system will not require any changes and can continue to operate as these components are evolved.



For example, a developer may wish to load an AI State Machine instead of the default Control State Machine that is used with standard Player character. This will allow the AI to perform actions that may be unique to them or to remove functionality that the players can use that the AI cannot. Additionally, as we update the components, such as the current Lua Animate script, the Animate component can be replaced by an Anim Graph component to allow seamless upgrades to the character system for developers.

Character Loader Package

The Character Loader package contains a **CharacterComponentLoaderClient** LocalScript, the **CharacterComponentLoaderServer** Script, the **ComponentInterfaceInstance** Module and the **CharacterHelper** instance. The **CharacterComponentLoaderClient** and **CharacterComponentLoaderServer** initiates the loading of the **ComponentInterfaceInstance** for player characters. This will load the **CharacterComponentInterface** module and create instances of the various modules within the shared CharacterComponents package. The **CharacterHelper** instance contains various C++ APIs to help accelerate certain behaviors of the character components.

Configurations

Configurations hold component specific static variables (ValueObjects) as well as signals that can be used to synchronized between clients and servers. Each component will have ownership over the

objects under a folder with the same name as the component. Components can register these initial configurations with the **CharacterComponentInterface**.

In the future, these may be moved to custom Attributes on the **CharacterComponentScript**. This will remove the need to break the package link in order for developers to modify the default configuration values.

CharacterComponentLoader (Client / Server)

This script starts the loading of the character components by creating an instance of the CharacterComponentInterface module. If a developer wishes to load the character components on an NPC, they can convert the client script to a server Script and use that inside the NPC to load all of the character components on the server.

ComponentInterfaceInstance

The ComponentInterfaceInstance loads an instance of the CharacterComponentInterface for this object and manages the access to this instance for other scripts and modules to use.

For examples on how to use this module, see the CharacterComponentInterface documentation below.

Common Character Components Packages

CharacterComponentInterface

The CharacterComponentInterface will facilitate the communication between the various components that can be used with the new character system. This module will maintain a set of properties that other components may wish to have to have access to. The various components, as their instances are created at load time, will register variables and functions with this interface to allow them to be accessed by other components and by developer scripts outside of the component structure. Additionally, values saved in the CharacterControlConfigurations can also be accessed through the same get and set interfaces as component variables.

Interfaces

External Interfaces

- **var GetVariable(VariableName, ...)** : If the requested variable has been registered by a component, the registered functions on the associated component will be called. Otherwise,

the internal value table of the interface will be checked for a value saved there. If neither of these locations can return a value for the requested variable, the call will return nil.

- **void SetVariable(VariableName, Value, ...)** : If the requested variable has been registered by a component, the registered functions on the associated component will be called. Otherwise, the internal value table of the interface will be set to the specified value.
- **Var CallFunction(FunctionName, ...)**: If the requested function has been registered by a component, the registered function will be called and the return values of that function returned as well. Otherwise, the function will return nil.

Internal Component Interfaces

- **void RegisterVariable(VariableName, GetFunction, SetFunction)** : Register a specific variable to be handled by the specified get and set functions. These functions should be of the form var GetFunction(self) or void SetFunction(self, value).
- **void LoadDefaultComponent(ModuleName, LoadBoolean)** : By default, all modules are loaded, but by calling this function with the module name and false for the LoadBoolean, the specified module will not be loaded. This function should be called before the matching Setup call to ensure that when the modules are loaded during the setup, only those that are desired are loaded.
- **void LoadUserComponent(ComponentModule)** : This adds a developer component to the character component loading list to allow developers to add custom components without modifying the CharacterComponentInterface script. The specified module will be loaded in the next **SetupClient** or **SetupServer** call. After all the user modules are loaded, the list of user components is cleared to prevent duplicate loading.
- **void SetupClient(CharacterModel, ServerOnly)** : This function sets the character model that these component instances are associated with and creates the component instances for all of the default component modules, unless directed not to load. The ServerOnly boolean lets the module know if this is a single data model character such as an NPC where both the client and server modules exist in the same server DataModel. For player controlled characters, this should be set to false for both Setup functions. For NPCs, this should be set to true to allow both sets of modules to coexist in the same DataModel.
- **void SetupServer(CharacterModel, ServerOnly)** : This function sets the character model that these component instances are associated with and creates the component instances for all of the default component modules, unless directed not to load. The ServerOnly boolean lets the module know if this is a single data model character such as an NPC where both the client and server modules exist in the same server DataModel. For player controlled characters, this should be set to false for both Setup functions. For NPCs, this should be set to true to allow both sets of modules to coexist in the same DataModel.

Registered Variables

- **MoveDirection:** Vector3 - The desired direction of the character from an input source.

- **MoveDirectionSource**: Component (read only) - A link to the component that has set the MoveDirection to detect when the input value has been overridden.
- **Jump**: bool - Indication if a jump is requested from input for this tick.

Examples

The CharacterComponentInterface acts as the main access point for components, providing a common interface for interactions between components and between external scripts and the character components. By calling the **GetVariable**, **SetVariable**, and **CallFunction** interfaces on the **CharacterComponentInterface** module, other scripts and modules can retrieve, set or otherwise invoke functionality that each of the components provide.

The following code can be used to get the current **ComponentInterfaceInstance** instance for the specified **character.**

```
local characterComponenScripts = character:FindFirstChild("CharacterComponentScripts")
local cci =  require(characterComponenScripts :FindFirstChild("ComponentInterfaceInstance"))
```

This instance can then be used to access all of the component functionality exposed by the components through the registered variables and functions. For example, to retrieve the character's current world velocity, it can be queried using the following code.

```
local WorldVelocity = cci:GetVariable("WorldVelocity")
```

Similarly, we can request the character to be rigged using the following code to invoke the **ScaleRig** component to scale and rig the character. Since the **ScaleRig** component registers the **BuildRigFromAttachments** function with the component interface, the calling code does not need to know which component is handling this functionality and if the developer wishes to override this function, they can modify the default component or create their own that registers the same function to provide that service to the code that wishes to invoke that function.

```
cci:CallFunction("BuildRigFromAttachments", true)
```

Template

If developers wish to create their own components, they can use the following template to create a basic component for use in the character. For a component that provides PC functionality on the client, this new component should live in ReplicatedStorage to ensure that all clients have access to the module. If the component provides PC functionality on the server or provides functionality that is intended to be used solely by NPCs on the server, the component should live in the ServerScriptService to allow it to be loaded on the server but prevent it from having to replicate to all clients.

```lua
local ComponentTemplate = {}
ComponentTemplate.__index = ComponentTemplate
ComponentTemplate.ModuleName = "ComponentTemplate"

function ComponentTemplate:TestFunction()
    -- put function code here
end

function ComponentTemplate:GetTestVariable(value, test)
    if self.testVariable < test then
        self.testVariable = value
    end
end

function ComponentTemplate:GetTestVariable()
    return self.testVariable
end

function ComponentTemplate.new(CharacterComponentInterface, ServerOnly)
    local self = {
        name = script.Name,
        characterComponentInterface = CharacterComponentInterface,
        characterModel = CharacterComponentInterface.characterModel,
        moduleConfigurationObject =

CharacterComponentInterface.ConfigurationsObject:FindFirstChild(ComponentTemplate.ModuleName
),
        serverOnly = ServerOnly,


        componentVariable = 1,
        testVariable = 3,
    }
    setmetatable(self, ComponentTemplate)

    -- Register Any Configurations Added
    CharacterComponentInterface:RegisterConfigurations(ComponentTemplate.ModuleName)

    -- Register Variables
    CharacterComponentInterface:RegisterVariable("ComponentVariable",
        CharacterComponentInterface:MapGetToVariableInModule(self, "componentVariable "),
        CharacterComponentInterface:ReadOnlySetFunction("componentVariable "))
    CharacterComponentInterface:RegisterVariable("TestVariable",
        CharacterComponentInterface:MapGetToGetFunctionInModule(self, "GetTestVariable"),
        CharacterComponentInterface:MapSetToSetFunctionInModule(self, "SetTestVariable"))

    -- Register Functions
    CharacterComponentInterface:RegisterFunction("TestFunction",
        CharacterComponentInterface:MapToFunctionInModule(self, "TestFunction"))
end

--Remove and cleanup events
function ComponentTemplate:delete()
end

return ComponentTemplate
```

BaseStateMachine

The basic state machine defines the common functionality that would be used by any state machine system. This includes constructors and destructor functions to create an instance of the state machine as well as functions to set the current state of the state machine and a function that is called for each step of the state machine.

Registered Variables

- **CurrentStateEnum (Enum.State):** Reference to the current state of the state machine
- **OnStateChanged(Event(string StateName)) :** BindableEvent that is fired after a state change occurs. The name of the new state will be passed at the first argument to any event handlers that are registered to this event

Registered Functions

- **void AddState(State)** : This will add the specified state from the character state machine. Transitions to this new state will still need to be added in order for the state machine to transition to this new state.
- **State GetState(StateName)** : This function will return the instance for the current state of the character state machine.
- **void RemoveState(State)** : This will remove the specified state from the character state machine, if it exists. This does not remove references by transitions to this state.

Registered Configurations

- **WalkSpeed** : (number) Max speed of the character in studs/sec
- **JumpHeight** : (number) Height of a standing jump for the character (in studs).
- **StepHeight** : (number) Height of steps that the character can walk up. Measured in studs.
- **MaxSlopeAngle** : (number) Maximum slope of floor that the character can walk up in degrees
- **RespawnTime** : (number) Number of seconds to spend in the death state before respawning the character
- **RespawnEvent** : (RemoteEvent) Triggered when the client wants to respawn the character. Only works on the owned PC.
- **SitEvent** : (RemoteEvent) Triggered when the client wants to the character to sit in a Seat. Only works on the owned PC.

Component Dependencies

- Heartbeat - AddHeartbeatCallback (Function)
- Heartbeat - RemoveHeartbeatCallback (Function)
- CharacterComponentInterface - MoveDirection (Variable)
- CharacterComponentInterface - Jump (Variable)
- Health - CurrentHealth (Variable)
- PhysicsComponent - MovementAcceleration (Variable)
- PhysicsComponent - TargetMovementVelocity (Variable)
- PhysicsComponent - TargetFacing (Variable)
- PhysicsComponent - IsGrounded (Variable)
- PhysicsComponent - IsBuoyant (Variable)
- PhysicsComponent - SurfaceObject (Variable)
- PhysicsComponent - ImpulseSpeed (Variable)
- PhysicsComponent - WorldCFrame (Variable)

BaseState

The State Machine is made up of various Character States. These states are derived from the BaseState class and implement the functionality to map user input to character controls and physics motion.

Interfaces

- **OnEnter**: const StateMachine(In) : A function that is called when the state is entered. This function will be called before the state OnHeartbeat is called. It can be used to set up the state before it begins.
- **OnHeartbeat**: const StateMachine(In) : A function called each heartbeat. This function can be used to read the UserInput and set the various properties of the Character Physics Controller to help drive the character.
- **OnExit**: const StateMachine(In) : A function that is called when the state is exited. This function will be called before the new state OnEnter is called. It can be used to clean up the state after it is complete.
- **PushTransition** : Transition(In) : The specified transition is added to the list of transitions for the current state. Transitions will be tested in reverse order that they are added (stack).
- **RemoveTransition** : String TransitionName(In) : The first transition with the specified name (if any) will be removed from this state.
- **GetName** : string (Out) : Get the name of the state

BaseTransition

The BaseTransition is a base implementation of a test if a transition from the current state to a new state.

Interfaces

- **SetToState**: State(In): Sets the state that will be the next state if this transition evaluates to true when tested
- **Test**: StateMachine(In): Implements the test condition for this transition. If, during the tick, this function returns true, the state machine will transition to the specified ToState
- **GetName** : string (Out) : Get the name of the transition

Default Character Components Packages

Animate (Client)

The Animation component works with the character rig and skeleton to play animations on the character. Animation retargeting will be supported in this component with in conjunction with the Character Appearance system. The animation state machine uses the generic state machine along with input from the Character State Machine and the Character Physics Controller.

Registered Configurations

- **List of Animations** : StringValue [Name], Animation(s), NumberValue [Weight]

Component Dependencies

- ControlStateMachine - OnStateChanged (Variable)
- ControlStateMachine - StepHeight (Variable)
- PhysicsComponent - WorldVelocity(Variable)
- Heartbeat - AddHeartbeatCallback (Function)
- Heartbeat- RemoveHeartbeatCallback (Function)

Appearance (Client)

Character Appearance controls the rendering and display of the character. This component will define what body parts the character has, how they are connected and how they are drawn on the screen. This component encompasses two major sets of functionality, both related to character appearance.

AutoMove (Client)

The Client Auto Move Module helps to supports character movement over time to allow characters to move from their current location to a specified location. This module would be used by click to move and by AI systems to support character movement.

NOTE: When MoveTo is used by a local script on the player, the player should be able to cancel the functionality by pressing any of the movement keys. As soon as input is read, the input essentially overrides the move to functionality. Jump does NOT override move to.

Registered Variables

- **OnMoveToFinished :** BindableEvent(bool reached) - This event fires when the AutoMoveModule move completes.
- **MoveTo :** Vector3 location , Instance part - Direct the character to move to the specified location in world or local space if part is specified. Get will return the location (Vector3), if any
- **MoveToPart** : Instance(Read Only) - Returns the current MoveTo part, if any

Registered Functions

- **StopMoveTo** : bool - A function that stops the character from moving, regardless of if the destination has been reached or not

Registered Configurations

- **DistanceEpsilon:** float - Used to determine if a character is "close enough" to the desired goal. Once the character is within range of the desired location, the character stops.
- **MoveTimeout**: float - This is a timeout that will stop the character if it doesn't reach its goal. This is done so that NPCs won’t get stuck waiting for MoveToFinished to fire. If what you don’t want this to happen, you should repeatedly call MoveTo so that the timeout will keep resetting or change the value.

Component Dependencies

- Heartbeat - AddHeartbeatCallback (Function)
- Heartbeat - RemoveHeartbeatCallback (Function)

- CharacterComponentInterface - MoveDirection (Variable)

BodyColor (Client)

The BodyColor component applies the values of the BodyColor to the parts that make up the character. If present, this component will read the color values on any BodyColor objects and apply those colors to the appropriate body parts. If the component cannot find a BodyColor object as a child of the character model, the component will not change the color of the character.

NOTE: There are no registered variables or functions in this module.

NOTE: There are no component dependencies in this module.

ControlStateMachine (Client / Server)

This state machine describes the various control states that a character can be in, such as on the ground, falling through the air, climbing ladders, swimming. Each state describes a unique set of controls or interactions that map to the behavior of the character.

States within the State Machine transform the User Input and forward it to the Character Physics Controller in terms of desired motions and velocities. This allows the states to choose how the player input will be mapped to allow different controls schemes based on the character's state.

This state machine also controls how the character can enter and leave each of these states, whether through interactions with the world, like falling off a platform, or through user input, such as hitting 'jump'. Characters will be in a single state at a time and transitions between states will happen instantaneously, with no time being spent in the transition between states. These transitions can query the Character Physics Controller, character sensors or other components as part of the transition checks to determine when to change states. For example, when the character is in the Ground state, if the state transition detects that the Character Physics Controller IsGrounded() is not true, the engine will transition to the Air state. Similarly, more complex state transition detectors, such as ladder detection, can be either implemented in Lua to allow developer customization or if performance becomes an issue, implemented in a custom C++ detector function that can be attached to the state machine transition.

In relation to the Avatar Motion Framework, this state machine provides the Intent functionality and works independently from the later stages of the movement flow. This allows these other stages to change their implementation and evolve without impacting this state machine.

Ideally, this component will allow developers to customize how their physical representation in the game behaves based on user input and physical responses and even specify their own state machines to control how their avatars will interact with the world.

A default implementation of the Character Control State Machine is provided to implement basic character functionality. The initial version will be built in Lua using a common State Machine framework with the option of adding specific tests to C++ to improve performance. This

implementation allows developers full visibility to the workings of the Character State Machine and allows them to edit, modify or replace any sections that they wish to best suit their game's needs.

Performance of the Character Control State Machine being in Lua should not be an issue. There are two types of Avatars that may show up in a game. PCs will be running locally on each client that belongs to the player and only one copy of the State Machine will be executing on any particular client. NPCs will be running on the server, but in most cases will not need the full functionality of a PC state machine and can be truncated and optimized to work well on the server.

Additionally, neither PC nor NPCs will be subject to transition between client and server. PC State Machines should only ever run on the client that is controlling them and NPC State Machines should only ever run on the server to prevent hacking and manipulation by malicious clients.

This implementation does not preclude implementation of a black box C++ implementation of a State Machine in the future if performance becomes a significant issue for particular use cases of the Character State Machine.

- Does not solve the issue of server authoritative PC movement and controls for competitive games

Character State Machine

Registered Variables

- **CurrentStateEnum** : Enum.HumanoidStateType - Gets the Humanoid state enum equivalent (if any) for the current state
- **OnStateChanged** : BindableEvent(string NewStateName) - Event that fires after a state transition has occurred

Default States and Transitions

States within the State Machine transform the User Input and forward it to the Character Physics Controller. This allows the states to choose how the player input will be mapped to allow different controls schemes based on the character's state.

The standard implementation of the control state machine will contain the following states by default and these states are designed to mimic the behavior of the current Humanoid.

Developers will have the ability to disable any of the default states by removing all transitions to that state. Additionally, they will be able to create their own states and transitions and add them to the default state machine without having to fork the implementation of the default state and transition code.

Ground

Ground state describes the control set when the character is standing on the ground. The input controls are mapped to the standard forward, back, left, and right which move the character in their respective directions along the plane of the ground.

- **Jump**: Jump control input is also active and transitions to the Jump state.
- **Air**: If a character falls off a surface so that there is no ground beneath them, the character will transition to the Air State.
- **Ladder**: If a ladder is detected (by the helper function) in front of the character, the character transitions to the Ladder state.
- **Water**: If the character detects that is now immersed in water, the character will transition to the water state.
- **Sit**: If the character activates a seat, the character will transition to the Sit state.
- **Dead**: If the character runs out of health, the character transitions to the Dead state.

Air

Air state specifies the controls when the character is in the air (not on the ground). While in the air, the standard are mapped to the standard forward, back, left, and right which move the character in their respective directions perpendicular to the up vector. The amount of control while in the air is less than while on the ground.

- **Ground**: If the ground is detected under the character, the state is changed to Ground.
- **Ladder**: If a ladder is detected (by the helper function) in front of the character, the character transitions to the Ladder state.
- **Water**: If the character detects that is now immersed in water, the character will transition to the water state.
- **Dead**: If the character runs out of health, the character transitions to the Dead state.

Jump

Jump is a transitional state that applies an upward force to the character and then transitions to the Air state.

- **Dead**: If the character runs out of health, the character transitions to the Dead state.
- **Air** : Otherwise, the character immediately transitions to the Air state.

Swim

Swim state specifies the controls while the character is in water. The input controls are mapped to the standard forward, back, left, and right which move the character in their respective directions perpendicular to the up vector. While under the surface of the water, the jump input is mapped to the up direction.

- **Jump**: Jump control input is also active and if the character is on the surface of the water, it transitions to the Jump state.
- **Air**: If a character moves out of the water so they are no longer buoyant, the character will transition to the Air State.
- **Dead**: If the character runs out of health, the character transitions to the Dead state.

Ladder

The ladder state controls character behavior while they are climbing a linear ladder. This state maps the forward and back controls to move the character up and down the ladder respectively.

- **Jump**: Jump control input is also active and transitions to the Jump state.
- **Air**: If a character falls off a surface so that there is no ladder in front of them, the character will transition to the Air State.
- **Ground**: If the ground is detected under the character and there is no ladder in front of them, the state is changed to Ground.
- **Dead**: If the character runs out of health, the character transitions to the Dead state.

Sit

This state is active while the character is sitting in a Seat or Vehicle seat. All standard directional controls do not move the character while in the Sit state. Directional controls may be mapped through the controls of the VehicleSeat if the character is sitting in one.

- **Jump**: Jump control input is also active and transitions to the Jump state.
- **Dead**: If the character runs out of health, the character transitions to the Dead state.

Ragdoll

Ragdoll is a generic state the disables all of the standard controls and can be used by the Physics component (or others) to create alternative physical models such as ragdoll.

- **Ground**: After the respawn timer, the character is transitioned back to the Ground state.
- **Dead**: If the character runs out of health, the character transitions to the Dead state.

Dead

Dead state is used when a character has no more health. The state the disables all of the standard controls and will transition out of this state automatically .

- **Ground**: After the respawn timer, the character is transitioned back to the Ground state.

Equipment(Client / Server)

This module manages the equipping and unequipping of accessories. Additionally, if an accessory parented to a character with this module, the item will automatically be added and equipped to the character. If an added accessory is unparented from the character, it will be automatically removed or unequipped from the character.

NOTE: There are no component dependencies in this module.

RegisteredFunctions

- **AddItem (Instance Accessory, bool Add) :** attaches or removes the specified Accessory to the character depending on the passed bool value.
- **List<Instance> GetAccessoryList() :** Returns a list of all current attached accessories.

Health (Client / Server)

The Health script maintains the character's current health and provides inputs to adjust the character's health. Additionally, this script provides display functionality should the developer wish the character's health to be visible in the game world.

Registered Configurations

- **CurrentHealth :** float - Current health of the character
- **MaxHealth** : float - Maximum health of the character
- **HealthRegenRate** : float - Rate of health regen in health per sec

Component Dependencies

- Heartbeat - AddHeartbeatCallback (Function)
- Heartbeat - RemoveHeartbeatCallback (Function)

Heartbeat (Client / Server)

The Heartbeat component manages component callbacks that are run each engine step. This component allows there to be a single callback registered with the engine to provide all the callback

calls from all registered components across all character component in the current DataModel. This provides significant performance improvements and allows a single place to find all regularly called component Lua code.

This component can call functions during the following times:

- **heartbeat**: Called after the physics step of this engine tick
- **stepped** : Called before the physics step of this engine tick
- **earlyStepped**: Called before the physics step and before the stepped stage

RegisteredFunctions

- **AddHeartbeatCallback(function Callback, Component ComponentInstance, String HeartbeatType, Number Priority) :** Adds the passed callback to the list of requested heartbeat callbacks for the component. This callback will be run during the specified HeartbeatType at the specified priority.
- **RemoveHeartbeatCallback(Component ComponentInstance, String HeartbeatType) :** Removes all callbacks (if any) in the list of requested heartbeat callbacks for the specified Heartbeat type for this component instance.

PhysicsController (Client / Server)

The **PhysicsController** component provides all of the simulation that controls how the character root part reacts to input and the physical aspects of the game world. The **PhysicsController** should be state agnostic and should not depend on what the current state of the character is to determine how it reacts to the world. Any required changes in behavior based on state should be controlled by the inputs to the Physics Controller.

Character physical representation may change as needed by the game. For example, different **PhysicsControllers** may have physical representation such as the current **HumanoidRootPart** or a capsule / ray based system. These changes could occur at run time based on character state, such as crawling or crouching. Additionally, there may be various LODs of the physical representation based on proximity to certain objects or needs of the game.

Currently, the server side component creates the required objects to set up the **CharacterPhysicsController** and when shut down, it removes any objects created as part of the controller. The client component registers a callback on the heartbeat to handle moment to moment physical changes and behaviors and on shutdown, it unregisters any event bindings that may have been made to support it.

NOTE: There are no component dependencies in this module.

Registered Variables - Client

- **TargetMovementVelocity:** Vector3 - Desired velocity based on player input to move the character root object.
- **MovementAcceleration**: Vector3 (In) - Desired weight modifier to acceleration to desired movement
- **TargetFacing**: Vector3 - Desired world facing of character
- **ImpulseSpeed** : Vector3 - Any additional forces being applied to the character based on states or other input factors. This will be multiplied by the mass of the character to derive the required impulse.
- **WorldCFrame**: CFrame (ReadOnly) - World Position / Orientation of character controller
- **WorldVelocity** : Vector3 (ReadOnly) - The resulting current velocity of the character after physical simulation is applied.
- **IsBuoyant** : Bool (ReadOnly) - Is the character in water?
- **OnWaterSurface**: Bool (ReadOnly) - Is the character on the surface of the water?
- **Buoyancy** : Vector3 (ReadOnly) - Any resulting forces due to fluid buoyancy that are currently being applied to the character.
- **IsGrounded** : Bool (ReadOnly) - Is the character on a surface?
- **SurfaceObject** : Object (ReadOnly) - The current object that the character is currently resting on (if any).
- **SurfaceVelocity** : Vector3 (ReadOnly) - The resulting current velocity of the character after physical simulation is applied with respect to the current surface (if any)
- **SurfaceNormal** : Vector3 (ReadOnly) - The surface normal that the character is currently resting on (if any)
- **SurfaceMaterial** : Material (ReadOnly) - The surface material of the object face that the character is currently resting on (if any)
- (Possible Future) : Current Contacts (List) - List of objects that are currently contacting the character and the amount of force that they are applying as a result of the contact

Component Dependencies

- ControlStateMachine - StepHeight (Variable)

PlateDisplay (Client)

The UI component displays the character's name. Currently, this functionality is implemented in C++, but the new component will include a Lua implementation of the character display that will let developers customize the way their character's name and other game information appears in the game.

RigScale (Server)

This module controls the scaling functionality of the character, including any attachments and accessories as well as rigging. It is currently implemented so that scaling is only possible from the server. Any changes done from the client will be ignored.

NOTE: This module runs on the server side. However, this script can be moved to the ServerRigScale folder and used from the client side (must include ServerRigScale to the list of client modules from ServerRigScale).

Registered Functions

- **BuildRigFromAttachments() :** bool - Hacky way of calling a function at the moment. If the boolean that is passed in is true, then this method builds the rig for the character from the current attachments.

Registered Configurations

- **BodyDepthScale :** number value - Controls the z-axis scaling of the character
- **BodyHeightScale :** number value - Controls the y-axis scaling of the character
- **BodyProportionScale :** number value - Controls wide/narrow properties of the character
- **BodyTypeScale :** number value - Controls R15 to anthro scaling of the character
- **BodyWidthScale :** number value - Controls the x-axis scaling of the character
- **HeadScale :** number value - Controls the scaling of the head of the character

Component Dependencies

- ControlStateMachine - StepHeight (Variable)

Sound (Client / Server)

The Sound component sets up the required sound scripts in the character to allow the character to play sounds correctly.

Component Dependencies

- ControlStateMachine - OnStateChanged (Variable)
- ControlStateMachine - GetState (Function)
- ControlStateMachine - CurrentStateEnum (Variable)
- Heartbeat - AddHeartbeatCallback (Function)
- Heartbeat - RemoveHeartbeatCallback (Function)