How To Build Vehicles

BigWorld Technology 2.0. Released 2010.

Software designed and built in Australia by BigWorld.

Level 2, Wentworth Park Grandstand, Wattle St
Glebe NSW 2037, Australia
www.bigworldtech.com

Table of Contents

Chapter 1. Introduction

Vehicles are entities designed to be piloted and/or ridden by avatars. This document will guide you through the steps necessary to implement vehicles using BigWorld Technology.

Note

Examples of vehicles would be:

- A car or a horse that can be controlled by the player.
- A large boat that many avatars can board.
- A moving platform (often useful in puzzle games) that can move vertically (a lift), horizontally, or in any other direction.

This document makes frequent references to an example vehicle included in FantasyDemo, called Ripper. It is a pilotable hover bike vehicle that can carry only the pilot.

FantasyDemo's Ripper vehicle with Ranger on board

Chapter 2. Controlling Vehicles

This document assumes that the majority of vehicles will be directly controllable by the player. This requires reversing the direction of vehicle updates, and diverting user inputs toward the vehicle.

2.1. `controlledBy`

The `controlledBy` property sets from whom updates to the vehicle's position will come from.

Setting the vehicle's controller to the player gives responsibility for the vehicle's movement to the player's client.

```
# Give control of the Ripper to the client. At this point the
# client Ripper will get a .physics member.
self.controlledBy = avatar.base
```

`cell/Ripper.py`

In the example above, `controlledBy` is set to the player's client, thus giving it the ability to dictate the position of the vehicle. Setting `controlledBy` back to `None` returns control to the vehicle's cell.

2.2. `BigWorld.player( <entity> )`

In order to allow the vehicle to respond to user inputs, you can make the vehicle become the player.

To do this, you will need to implement a specialisation of the vehicle's client entity (in the same file) with the name `Player<entity name>`. For example, a client entity type of `Avatar` would require a corresponding `PlayerAvatar`.

Setting the player:

```
BigWorld.player( <entity_to_become_player> )
```

Changing the player as above will trigger the become player and become non-player events in the appropriate entities. You can use these callbacks to configure and later cleanup the user input handling mechanisms.

```
class PlayerRipper( Ripper ):
    def onBecomePlayer( self ):
        self.setupKeyBindings()
        self.filter = BigWorld.PlayerAvatarFilter()
        FantasyDemo.cameraType( keys.FLEXI_CAM )
        ...

    def onBecomeNonPlayer( self ):
        self.pilotAvatar = None
        self.filter = BigWorld.DumbFilter()
        self.keyBindings = []
```

`client/Ripper.py`

Note

Be careful when changing from a player vehicle to a non-player vehicle from within a player vehicle's member function. Changing the player back to the `Avatar` will cause the type of the vehicle to change immediately.

Chapter 3. Boarding and Alighting Vehicles

Boarding and alighting changes whether and entity's position is known relative to the world or relative to another entity such as a vehicle. When one entity boards another, it moves with the vehicle that it has boarded. This is most useful for vehicles where the entities are able to move around, such as the deck of a ship, but can also be useful when implementing vehicles where the avatar sits in a specific seat.

3.1. Functions

3.1.1. `Entity.boardVehicle()`

Boarding a vehicle makes the entity its passenger, and subject to its movements. Which vehicle the entity has boarded can be found through the entity's `vehicle` property, available on both the client and cell.

```
def requestBoardVehicle( self, sourceVehicleID ):
    vehicle = BigWorld.entities[ sourceVehicleID ]
    self.boardVehicle( sourceVehicleID )
    vehicle.passengerBoarded( self.id )
```

`cell/Avatar.py`

The *`vehicleID`* parameter is the entity ID of the vehicle that you are boarding (it cannot be `None` or `0`).

This function changes the space of the entity to that of the vehicle, and sets the entity's vehicle property.

3.1.2. `Entity.alightVehicle()`

Having an entity alight from its vehicle makes its movement relative to the world.

```
def requestAlightVehicle( self, sourceVehicleID ):
    vehicle = BigWorld.entities[ sourceVehicleID ]
    if self.vehicle != None:
        self.alightVehicle()
    vehicle.passengerAlighted( self.id )
```

`cell/Avatar.py`

This function changes the space of the entity from that of the vehicle to that of the world.

It also sets the entity's `vehicle` property to `None`. Do not call this function if the entity is not aboard a vehicle.

Note

Both `boardVehicle()` and `alightVehicle()` must be used only from within member functions of the entity's cell. This is because they require the real entity, not a ghost of an entity that is in reality in a different cell.

3.2. Apply gravity and moving platforms

It is important to remember to disable falling when using these functions.

Falling objects automatically board and alight based on whether they are standing on the world, in mid-air, or are standing on a `PyModelObstacle` marked with a `vehicleID`. This mechanism will override whatever value you set, therefore the need to disable falling.

```
self.pilotAvatar.physics.fall = False
```

client/Ripper.py

3.3. Player vehicle

To become the player, the vehicle requires a player specialisation of the vehicle. The type of the vehicle is changed dynamically to and from its player variation.

```
import BigWorld

class Ripper( BigWorld.Entity ):
    def __init__( self ):
        BigWorld.Entity.__init__( self )

    def enterWorld( self ):
        pass

    def leaveWorld( self ):
        pass

class PlayerRipper( Ripper ):
    def onBecomePlayer( self ):
        pass

    def onBecomeNonPlayer( self ):
        pass

    def handleKeyEvent( self, isDown, key, mods ):
        return True
```

Outline of client/Ripper.py

Chapter 4. Models and Animation

4.1. Pilot models

To display characters seated in or on the vehicle, use the entity's model as an attachment. Entities themselves cannot be used as attachments, so you will need to remove the model first, and then attach it to a hard point on the vehicle.

```
pilotModel = self.pilotAvatar.model    # hold the model with a local
self.pilotAvatar.model = None          # detach model from entity
self.model.mount = pilotModel          # attach model to hard point
```

For each seat in the vehicle, you will need a separate hard point, and a corresponding hard point in the avatar's model.

4.2. Animating the vehicle

Controlling the animations of passengers and vehicles is likely to be very vehicle-specific, and so might be implemented in script on a per-vehicle basis, rather than using the Action Matcher.

In the Ripper example, a simple Action Matcher is implemented using lists of single frame and transition animations. A method called during the hover vehicle physics tick, `PlayerRipper.checkAnims()`, manages the ripper's animation in flight.

```
vehicleActions = [ "RIdle",
                   "RTurnLeft",
                   "RTurnRight",
                   "Stop",
                   "Thrust" ]

 def checkAnims( self ):
    ...

    if self.wantTurn != 0:
        act = (self.wantTurn+1)/2 + 1
    elif self.wantMove != 0:
        act = (self.wantMove+1)/2 + 3
    else:
        act = 0

    act = int(act)

    ripperAction = self.model.action(Ripper.vehicleActions[act])
    ripperAction()
   ...
```

`client/Ripper.py`

Chapter 5. Asynchronous and Ghost Cell Considerations

5.1. Entering the world

When using the board / alight vehicle mechanism, the passenger's position is defined in terms of the vehicle. As a result, passengers will always enter the world after the vehicle. To allow the vehicle to correctly handle this, we require some form of notification of the passenger's entry into the world.

The excerpt below illustrates how to achieve this:

```
# This function is called by avatar when it enters the world.
def passengerEnterWorld( self, pilot ):
    pilot.model.visible = False
    pilot.model.motors[0].entityCollision = False
    if self.model.mount == None:
        pilotModel = pilot.model
        pilotModel.tracker = None
        pilot.model = makeModel( pilot.modelNumber, None )
        pilot.model.visible = False
        self.model.mount = pilotModel
        self.model.mount.RipperPilotIdle()
        self.model.mount.visible = True
        pilot.targetCaps = []
```

client/Ripper.py

5.2. Observing avatars interacting with vehicles

Communication with third parties is done via the vehicle's cell. To do this, have the vehicle cell broadcast any state-changing responses to `self.allClients`, with the ID of the affected avatar as a parameter.

```
def passengerBoarded( self, sourceAvatarID ):
    # Notify all clients about the Ripper being mounted.
    self.allClients.mountVehicle( True, sourceAvatarID )
```

client/Ripper.py

Remember that third parties might not have both the vehicle and its passengers in their area of interest, and so the ID might not be immediately resolvable into an entity. To handle this situation, the avatar should notify vehicles they are riding when they themselves enter the world. The passenger will never enter the world before their vehicle, as its position is known only relative to the former.

Chapter 6. Ripper Example

The Ripper is a simple single-passenger hover bike, designed to demonstrate vehicle implementation. In this section, Ripper's implementation is detailed to help in its use as a basis for new vehicles.

The Ripper consists of the following files:

- `fantasydemo/res/scripts/base/Ripper.py`
- `fantasydemo/res/scripts/cell/Ripper.py`
- `fantasydemo/res/scripts/client/Ripper.py`
- `fantasydemo/res/scripts/entity_defs/Ripper.def`
- `fantasydemo/res/scripts/editor/Ripper.py`

It also requires an entry in `fantasydemo/res/scripts/entities.xml` file, and some functionality in the Avatar's client and cell.

6.1. Base Implementation

The Ripper's base is empty, as it does not need to do more than is provided by the default base of FantasyDemo.

6.2. Cell Implementation

The Ripper's cell manages control of the Ripper, and is responsible for notifying all interested clients of such changes.

When the Ripper is first created, the cell has control of its movements, but passes this ability to clients as they attempt to use it.

The Ripper's cell contains the methods described in the list below:

- **`mountVehicle( self, sourceAvatarID )`**

  Mount process stage 4 of 8, this is a Ripper specified method for mounting the vehicle. In this method, control is transferred to the client, and the specified avatar's cell is requested to board the Ripper. The board request of the avatar forms stage 5 of 8 of the dismount process.

- **`passengerBoarded( self, sourceAvatarID )`**

  Mount process stage 6 of 8, this is a callback from the pilot's cell, and is specific to the Avatar implementation

- **`dismountVehicle( self, sourceAvatarID )`**

  Dismount process stage 3 of 7, this is a Ripper-specific method for dismounting the vehicle In this method, the Ripper's cell takes back control of its movement, and requests that the avatar alight. The avatar's alight request forms stage 4 of 7 of the dismount process.

- **`passengerAlighted( self, sourceAvatarID )`**

  Dismount process stage 5 of 7, this is a callback from the pilot's cell, and is specific to the Avatar implementation.

- **`onLoseControlledBy( self, id )`**

This is a server event handler that should be implemented by all vehicles that transfer control to a client at some point. It is called when the connection to a passenger's client is lost, and so gives the vehicle the opportunity to return to a good state

6.3. Client implementation

The client side of the Ripper is the largest part of the implementation, as it handles interaction and animation.

Ripper's client contains the methods described in the list below:

- **`enterWorld( self )`**

  Implemented by most entities, for the Ripper contains cosmetic code that does not affect functionality. Particle systems, shadows, and the initial state of animation is set up. The passenger will always enter the world after the vehicle, requiring a separate `onPassengerEnterWorld()`.

- **`prerequisites( self )`**

  Part of the asynchronous loading system employed by the BigWorld client. By returning a list of assets used by the Ripper, pauses from blocking loads can be avoided.

- **`passengerEnterWorld( self, pilot )`**

  Called by the Avatar implementation in FantasyDemo when an Avatar enters the world on a vehicle. It is used by the Ripper to setup the pilot model and animation.

- **`leaveWorld( self )`**

  Implemented by most entities, for the Ripper contains only some minor cleanup code for particle systems and shadows. The client caches entities that leave the area of interest, so be sure that the vehicle will return to a good state if `enterWorld()` is called on it later.

- **`use( self )`**

  Mount process stage 1 of 8, this is an event handler override for when the entity is used. In the case of the Ripper, it is used to initiate the mounting process.

- **`walkToMountPosition( self, success )`**

  Mount process stage 2 of 8, this recursive function lines up the player and the Ripper for the mounting animations by using the `seek` commands in the player's physics object

- **`arriveAtMountPosition( self, success )`**

  Mount process stage 3 of 8, this callback is given to the `seek` function to notify the Ripper when the player has arrived. On success, the client-side Ripper will contact its cell and attempt to acquire control for this client.

- **`mountVehicle( self, succeeded, pilotAvatarID )`**

  Mount process stage 7 of 8, this is a broadcast sent to all clients interested in this Ripper. It informs them that an Avatar is mounting the vehicle — the affected Avatar also receives this call, and uses the ID parameter to identify itself. The Ripper becomes the player in this case, and plays the mounting animations to demonstrate this

- **`finishMountVehicle( self )`**

  Mount process stage 8 of 8, this is called at the end of the mount animation using the `BigWorld.callback()`[1] mechanism. It completes the mount process, unlocking the Ripper controls.

- **`dismountVehicle( self, pilotAvatarID )`**

  Dismount process stage 6 of 7, this is the second broadcast method used by the cell to notify clients of the Ripper's dismount in progress. As in `mountVehicle()`, the pilot ID is provided to identify the client dismounting.

- **`finishDismountVehicle( self )`**

  Dismount process stage 7 of 7. Completing the dismount animation, the player's model is returned to the avatar, which is then released back to user control.

6.3.1. PlayerRipper

PlayerRipper's client contains the methods described in the list below:

- **`onBecomePlayer( self )`**

  Event handler triggered when the vehicle becomes a PlayerRipper. Note that the PlayerRipper's constructor is not called, as the object is constructed as a normal Ripper and then later changes type to PlayerRipper.

- **`setupKeyBindings( self )`**

  Method used by the Ripper to setup the player's controls.

- **`onBecomeNonPlayer( self )`**

  Event triggered when the Ripper changes back into a plain Ripper object. Upon receiving this, clean up any members created during `onBecomePlayer` event, such as key bindings.

- **`handleKeyEvent( self, isDown, key, mods )`**

  Calls functions mapped to key inputs. This is basically the same as the one used in the Fantasydemo Avatar.

- **`updateThrust( self )`**

  Method called as part of the key event handling process to update the physics object and animations.

- **`beginDismount( self, isDown )`**

  Dismount process stage 1 of 7, this method shuts down the Ripper, so that it will fall to the ground, ready for the player to alight.

- **`dismountStep( self )`**

  Dismount process stage 2 of 7, this method finishes the landing initiated in `beginDismount`, before initiating the cell's dismount process.

- **`checkAnims( self )`**

  Method called to tick the animation of hover vehicles, as the generic action matcher is not suitable for this style of vehicle.

- **`playTransitionAction( self, act, oldTurn, oldMove )`**

  Utility function used by `checkAnims` to blend between different states of turning.

- **`handleConsoleInput( self, string )`**

  Method that passes its calls onto the avatar that handles the chat console.

- **`onCollide( self, newMomentum, collidePosition, severity, triangleFlags )`**

  Method used by the Rippers to trigger sounds and particle effects. The Ripper is a physical body in the world, and so can receive collision callbacks.

- **moveForward(self, isDown)**

  Key event handlers that translate the relevant 'key down' state into a numerical one or zero. For use in

- **moveBackward(self, isDown)**

  See the `moveForward( self, isDown )` entry above.

- **`turnLeft( self, isDown )`**

  See the `moveForward( self, isDown )` entry above.

- **`turnRight( self, isDown )`**

  See the `moveForward( self, isDown )` entry above.

- **`moveUpward( self, isDown )`**

  See the `moveForward( self, isDown )` entry above.

- **`updateThrust`**

  See the `moveForward( self, isDown )` entry above.

6.4. Ripper definition file

Located under `fantasydemo/res/scripts/entity_defs`, the `Ripper.def` definition file for the Ripper contains the following properties:

- **`pilotID`**

  Current pilot of the vehicle It is used in client-side checks to prevent multiple mount attempts, and also for displaying the pilot model on the vehicle.

The file declares the following client methods:

- **`mountVehicle()`**

  Function call sent to all interested clients, to notify them of an avatar boarding the Ripper.

- **`dismountVehicle()`**

  Function call sent to all interested clients, to notify them of an avatar dismounting the Ripper.

It also declares the following cell methods:

- **`mountVehicle()`**

  Function used by the client to actually mount the vehicle and gain control of it. This function will fail if the Ripper is already occupied.

- **`dismountVehicle()`**

  Function called by either the client or the controller lost event handler to return the Ripper to a mountable state.

- **`passengerBoarded()`**

  Callback received from the avatar when it has boarded the vehicle entity.

- **`passengerAlighted()`**

  Callback received from the avatar when it has alighted from the vehicle entity.

6.5. Ripper editor

Located under `fantasydemo/res/scripts/editor`, the purpose of the editor Ripper is to provide the world editor with a representative model that it can display.

For the Ripper, we just use its model.

6.6. Ripper requirements of Avatar

To work with vehicles, the Avatar entity must implement the client methods below:

- **`enterWorld( self )`**

  In order to display correctly on clients observing other Avatars riding Rippers, the Avatar must notify its vehicle when it enters the world, as this will be after the vehicle enters it. In FantasyDemo, this is done using the function `onPassengerEnterWorld()`.

It must also implement the following cell methods:

- **`requestBoardVehicle( self, sourceVehicleID )`**

  Because the entity board and alight functions cannot be called on a ghost cell, they need to be wrapped in ghost-safe functions defined in the definition file.

- **`requestAlightVehicle( self, sourceVehicleID )`**

  Similar to `requestBoardVehicle()` this provides a ghost-safe wrapper for the entity alight vehicle functionality. It also does some checking, to make it safe to call multiple times before remounting.

- **`chat()`**

  In FantasyDemo, the chat system is implemented through the avatar cell. Because of this, the Ripper assumes its implementation.