Article

Adding touch controls to games that support game controllers in iOS

Use touch input and virtual controllers to make your game available to players without controllers.

Overview

Many people play games on iOS devices without using a physical controller. You can make your game available to them by adding touch controls, even if you designed your game for a physical controller. The `GCVirtualController` class leverages the Game Controller framework and optionally provides a default user interface. `GCVirtualController` can also render custom controls.

Create a custom view controller subclass

A Metal-based game typically presents content in a `UIView`. An instance of `UIViewController`, known as the root view controller, manages the `UIView`. Your root view controller is responsible for the `GCVirtualController` instance. To take responsibility, subclass from `GCEventViewController` rather than `UIViewController`.

> **Note**
>
> If your game supports tvOS, you need to subclass from `GCEventViewController`.

To create a custom subclass of `UIViewController`, create a new file in your project using the Cocoa Touch Class template. In the “Subclass of” combination box, enter “GCEventViewController”. In the Language menu, choose Objective-C. Then, open the newly created header file and import the Game Controller framework.

If the game uses a storyboard, update your main scene to use a new view controller class derived from `GCEventViewController`. Alternatively, if the game creates a root view controller programmatically, then update the code to create an instance of a new subclass derived from `GCEventViewController`.

Configure the virtual controller

When you connect a `GCVirtualController`, a new `GCController` appears in the controllers array. The elements that this controller supports are controlled by the configuration created for `GCVirtualController`. To configure a virtual controller, follow these steps:

1. In your subclass of `UIViewController`, add a `GCVirtualController` instance variable.
2. Add an override to the `viewDidLoad()` method that creates the virtual controller and stores it in an instance variable.

```
@implementation MyGameViewController {
    GCVirtualController *_virtualController;
}

- (void)viewDidLoad {
    [super viewDidLoad];

    if (!_virtualController) {
        GCVirtualControllerConfiguration *config = [[GCVirtualControllerConfiguratio
        // Add elements that your controller-handling code currently supports.
        config.elements = [NSSet setWithArray:@[
            GCInputButtonA,
            GCInputButtonB,
            GCInputButtonX,
            GCInputButtonY,
            GCInputLeftThumbstick,
            GCInputRightThumbstick]];
        _virtualController = [[GCVirtualController alloc] initWithConfiguration:conf
    }
}
```

Show and hide the virtual controller

If there are no physical controllers connected, the game shows the virtual controller by default. If the player connects a physical controller, the game hides the virtual controller.

To show a virtual controller when there are no physical controllers connected, call `connect(replyHandler:)` on it in the `GCControllerDidDisconnect` (Swift) or `GCControllerDidDisconnectNotification` (Objective-C) handler.

```
- (void)controllerDidDisconnect:(NSNotification *)notification {
    if (GCController.controllers.count == 0 && _virtualController != nil) {
        [_virtualController connectWithReplyHandler:nil];
    }
}
```

When your game starts and there are no controllers connected, call `connect(replyHandler:)` on the virtual controller to show it.

```
- (void)gameDidStart {
    if (GCController.controllers.count == 0 && _virtualController != nil) {
        [_virtualController connectWithReplyHandler:nil];
    }
}
```

To hide the virtual controller when a player connects a physical controller, call disconnect on the virtual controller in the `GCControllerDidConnect` (Swift) or `GCControllerDidConnectNotification` (Objective-C) handler. The game might receive a connection notification for a physical controller before receiving a notification for the virtual controller because controller connections are asynchronous.

```
- (void)controllerDidConnect:(NSNotification *)notification {
    if (_virtualController != nil) {
        BOOL hasPhysicalController = NO;
        for (GCController *controller in GCController.controllers) {
            if (controller != _virtualController.controller) {
                hasPhysicalController = YES;
                break;
            }
        }
        if (hasPhysicalController) {
            [_virtualController disconnect];
        }
    }
}
```

Show and hide controls

When the player can't perform certain actions in the game, hide the corresponding control elements using the `updateConfiguration(forElement:configuration:)` method. For example, during cutscene playback, the player can only press a button to skip playback.

```
- (void)configureVirtualControllerForCutscene:(BOOL)isInCutscene {
    if (!_virtualController) {
        return;
    }

    // Only the X button is available when in a cutscene.
    NSArray *elementsToShowOrHide = @[
        GCInputButtonA,
        GCInputButtonB,
        GCInputButtonY,
        GCInputLeftThumbstick,
        GCInputRightThumbstick];
    for (GCInputElementName input in elementsToShowOrHide) {
        [_virtualController updateConfigurationForElement:input
                                    configuration:^(GCVirtualControllerElementConfiguration
            configuration.hidden = isInCutscene;
            return configuration;
        }];
    }
}
```

Render custom touch controls

For better immersion and performance, the game can render a custom virtual controller instead of the default. Custom controls render symbols that represent their mapped actions rather than generic symbols.

> **Note**
>
> When rendering custom touch controls, UIKit handles touch events and draws the results as part of the game's user interface render pass. If your game is multithreaded, you need to pass data to your render thread in a thread-safe way because UIKit is only available on the main thread.

Set the hidden property to true when you create the GCVirtualController.Configuration. Continue to set the elements property because this affects the elements of the corresponding GCController.

```
- (void)viewDidLoad {
    [super viewDidLoad];

    if (!_virtualController) {
        GCVirtualControllerConfiguration *configuration = [[GCVirtualControllerConfi
        // Add elements that your controller-handling code currently supports.
        configuration.elements = [NSSet setWithArray:@[
            GCInputButtonA,
            GCInputButtonB,
            GCInputButtonX,
            GCInputButtonY,
            GCInputLeftThumbstick,
            GCInputRightThumbstick]];
        configuration.hidden = YES;
        _virtualController = [[GCVirtualController alloc] initWithConfiguration:conf
    }
}
```

Add code to the implementation of viewDidLoad() to install one or more UIGesture Recognizer objects on the game's view. For example, use a UIPanGestureRecognizer to implement a direction pad or a UILongPressGestureRecognizer to implement a button. Set the view controller as the delegate and the target of each gesture recognizer, and store them in instance variables.

```
@implementation MyGameViewController {
    GCVirtualController *_virtualController;
    UIPanGestureRecognizer *_thumbstickGR;
    UILongPressGestureRecognizer *_primaryButtonGR;
}

- (void)viewDidLoad {
    // ...
    // ... Continued from above.
    // ...

    _thumbstickGR = [[UIPanGestureRecognizer alloc] init];
    _thumbstickGR.delegate = self;
    [_thumbstickGR addTarget:self action:@selector(thumbstickAction)];
```

```
    [self.view addGestureRecognizer:_thumbstickGR];

    _primaryButtonGR = [[UILongPressGestureRecognizer alloc] init];
    _primaryButtonGR.minimumPressDuration = 0; // Invoke action method immediately.
    _primaryButtonGR.delegate = self;
    [_primaryButtonGR addTarget:self action:@selector(primaryButtonAction)];
    [self.view addGestureRecognizer:_primaryButtonGR];
}
```

Then, declare an extension of the root view controller subclass that conforms to `UIGestureRecognizerDelegate`. In the extension, declare the action methods that the configured gesture recognizers invoke.

```
// Declares an extension (unnamed category) of MyGameViewController.
// Place this declaration in the implementation file instead of the header file.
@interface MyGameViewController() <UIGestureRecognizerDelegate>
- (void)thumbstickAction;
- (void)primaryButtonAction;
@end
```

Implement the action methods to interpret the location of the touch, update the virtual controller, and inform the render thread.

> **Note**
>
> Update your user interface from the main thread, instead of game controller code, to avoid unnecessary input latency.

```
- (void)thumbstickAction {
    UIGestureRecognizerState state = _thumbstickGR.state;
    if (state == UIGestureRecognizerStateBegan || state == UIGestureRecognizerStateC
        CGPoint translation = [_thumbstickGR translationInView:self.view];
        CGRect bounds = [self calculateThumbstickRegionBounds];
        translation.x /= (bounds.size.width / 2);
        translation.y /= (bounds.size.height / 2);
        [_virtualController setPosition:translation forDirectionPadElement:GCInputLe
        uiSystem->set_thumbstick_position_and_visibility(translation, true);
    } else {
        uiSystem->set_thumbstick_position_and_visibility(CGPointZero, false);
    }
}
```

```
- (void)primaryButtonAction {
    UIGestureRecognizerState state = _primaryButtonGR.state;
    if (state == UIGestureRecognizerStateBegan || state == UIGestureRecognizerStateC
        [_virtualController setValue:1.0 forButtonElement:GCInputButtonX];
        uiSystem->set_primary_button_pressed(true);
    } else {
        [_virtualController setValue:0.0 forButtonElement:GCInputButtonX];
        uiSystem->set_primary_button_pressed(false);
    }
}
```

Next, implement the `gestureRecognizer(_:shouldReceive:)` delegate method to define the bounds of the gesture recognizer. Calculate the bounding boxes of the controls based on the view's current size, and place it within the view's current safe area.

```
- (CGRect)calculateThumbstickRegionBounds {
    CGRect viewBounds = self.view.bounds;
    UIEdgeInsets safeAreaInsets = self.view.safeAreaInsets;
    CGSize thumbstickRegionSize = CGSizeMake(200, 200);
    return CGRectMake(
        safeAreaInsets.left,
        viewBounds.size.height - thumbstickRegionSize.height - safeAreaInsets.bottom
        thumbstickRegionSize.width,
        thumbstickRegionSize.height);
}

- (CGRect)calculatePrimaryActionButtonBounds {
    CGRect viewBounds = self.view.bounds;
    UIEdgeInsets safeAreaInsets = self.view.safeAreaInsets;
    CGSize actionButtonSize = CGSizeMake(88, 88);
    return CGRectMake(
        viewBounds.size.width - actionButtonSize.width - safeAreaInsets.right,
        viewBounds.size.height - actionButtonSize.height - safeAreaInsets.bottom,
        actionButtonSize.width,
        actionButtonSize.height);
}

- (BOOL)gestureRecognizer:(UIGestureRecognizer *)gr shouldReceiveTouch:(UITouch *)to
    // Constrain the gesture recognizer to tracking one touch at any given time.
    if (gr.numberOfTouches >= 1) {
        return NO;
```

```
    }

    CGPoint location = [touch locationInView:self.view];
    if (gr == _thumbstickGR) {
        CGRect thumbstickRegionBounds = [self calculateThumbstickRegionBounds];
        return CGRectContainsPoint(thumbstickRegionBounds, location);
    } else if (gr == _primaryButtonGR) {
        CGRect primaryActionButtonBounds = [self calculatePrimaryActionButtonBounds]
        return CGRectContainsPoint(primaryActionButtonBounds, location);
    }

    return YES;
}
```

Update GCControllerDidConnect (Swift) and GCControllerDidDisconnect (Swift) handlers to show or hide the custom user interface when the virtual controller is connected. For Objective-C, update the GCControllerDidConnectNotification and GCControllerDidDisconnectNotification handlers.

```
- (void)gameDidStart {
    if (GCController.controllers.count == 0 && _virtualController != nil) {
        [_virtualController connectWithReplyHandler:nil];
        uiSystem->set_visible(true);
    }
}

- (void)controllerDidConnect:(NSNotification *)note {
    if (_virtualController != nil) {
        BOOL hasPhysicalController = NO;
        for (GCController *controller in GCController.controllers) {
            if (controller != _virtualController.controller) {
                hasPhysicalController = YES;
                break;
            }
        }
        if (hasPhysicalController) {
            [_virtualController disconnect];
            uiSystem->set_visible(false);
        }
    }
}

- (void)controllerDidDisconnect:(NSNotification *)note {
```

```
    if (GCController.controllers.count == 0 && _virtualController != nil) {
        [_virtualController connectWithReplyHandler:nil];
        uiSystem->set_visible(true);
    }
}
```

See Also

Touch controller

`class` GCVirtualController

A software emulation of a real controller that you configure specifically for your game.