Mobile Game Development 1 Coursework: Code Explanation

McGhee, Ruari

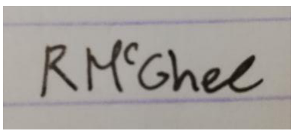
S1432402

GitHub repository located at: *https://github.com/Ruari026/MobileDev_Coursework*

I confirm that the code contained in this file (other than that provided or authorised) is all my own work and has not been submitted elsewhere in fulfilment of this or any other award.

Signature.

- Ruari McGhee



CONTENTS

ENGINE CLASSES

THE BASE BUILDING BLOCKS THAT THE GAME IS BUILT FROM

MAIN GAME

CLASS DESCRIPTION

The entry point of the game program, responsible for setting up the base that the program runs from.

DETAILED EXPLANATION

GAME INITALISATION

Since the loading of GameObjects is mostly handled by the individual scene's the Main Game OnLoad function's main point is to detect which platform that the game is being played on. This is important for handling the input of the game as if it is being played on mobile then the game needs to respond to screen touches whereas if the game is on browser then it needs to check for mouse clicks.

GAME LOOP

When handling the game loop most of the work is handled by the individual scene's the main purpose of the base run function is to ensure that the canvas that the game render's to is cleared before every scene update and render occurs and to calculate the amount of time that has passed since the game's last frame (the deltaTime). This is done so that any animations in the game and any movement can be scaled against real world time thus making these animations and movements consistent across all devices and platforms.

GAMESCENE

CLASS DESCRIPTION

A container for the full list of GameObjects that a game level needs to run, responsible for creating these objects and then ensuring that they are always updated and rendered each frame during the lifetime of the scene.

DETAILED EXPLANATION

GAMEOBJECT CREATION & SCENE STARTING

Any GameObjects that the scene requires to run tends to happen in the scene's LoadScene function. After the scene has been fully created all the GameObjects in the scene have an OnStart function called which allows them to handle any specific "interconnected" behaviour that they or their components may need to function.

SCENE UPDATING & RENDERING

Scenes delegates the updating and rendering behaviour to the components that the scene GameObjects have however they have finer control over the ordering of these events. Specifically, the scenes ensure that all of the objects Update first before rendering so that any objects that move and/ or animate has this handled first before showing the results to the screen.

The scene also ensures that if there are any objects that want to be added to or destroyed from the scene, that this happens after the full update and render loop has happened. This is so that there are no issues with array accessing/ array out of bounds issues that might occur if this were to happen mid update loop.

GAMEOBJECT

CLASS DESCRIPTION

The individual entities that make up a game level. Responsible for storing details such as the entity's position and related size details (for UI elements there are also anchor values to easily set the GameObject relative to the 4 corners or center of the screen which is handy when the game is scaled)

DETAILED EXPLANATION

PARENT CHILD RELATIONSHIPS

Instead of existing at a scene's root (the GameObject's position is relative to the world origin 0,0), GameObjects can be set to be a child of another GameObject (the GameObject's position is relative to its parent's position). This means that the GameObject can now have its position represented in two forms: its global position and its local position.

A GameObject's global position is calculated by getting the sum of its local position and its parent's global position. This represents the GameObject's total position in world space, an object's local position just represents the offset that the gameobject has relative to its parent (If the GameObject does not have a parent then its global position is the same as its local position).

This ability to "Connect" objects together means that if the parent gameobject moves then all of its connected child objects will also move with it so that they keep the same offset that they had before. However, since these children no longer exist at the scene's root (and the scene does not know about these GameObjects) it is the responsibility of the parent GameObject to ensure that the children update and render after the parent has updated and rendered.

This also has added the opportunity for large sets of GameObjects to be disabled and reenabled all at the same time as if you disable the parent GameObject the children of that GameObject will no longer Update and Render (this is particularly useful in menus for only showing/ hiding the relevant UI elements).

COMPONENT

CLASS DESCRIPTION

A blank slate to be extended from so that the various behaviour that GameObject's need for the game to run can be separated out and easily reused.

DETAILED EXPLANATION

METHOD OVERRIDDING

By itself, the component class does not do much as the behaviour that the GameObjects will need will be defined in various Subclasses. The base component class just gives 5 main methods that the subclassed components can override if they need.

- OnStart which is called after a scene has loaded but before the first update loop. This is intended to enable the component handled any complex setup that it may need to function properly (e.g., finding another gameobject or component in the scene that could not have been done during scene construction)
- Update which lets the gameobject have behaviour handled every frame.
- Input which is called any time the game receives an input event. The component will need to check the input event to see if it is relevant to its behaviour (also when checking the component should check for both mobile input and its associated web input just to ensure consistent behaviour across platforms)
- Draw which should be used when wanting to render anything to the screen, like Update this is called every frame however the scene ensures that this gets called after all the GameObjects in the scene have been updated (the timing is important between the two)
- End which can be used to handle any specific behaviour that may need to occur when the GameObject is being destroyed.

COMPONENTS

CLASSES THAT SUBCLASS FROM COMPONENT TO ENABLE BEHAVIOUR ON GAMEOBJECTS

AUDIO SOURCE

: SUBCLASSES FROM COMPONENT

CLASS DESCRIPTION

An Audio Source is a Component that is responsible for handling the playing of sound effects and background music. Depending on the platform that the game is being played on (Web vs Mobile) the Audio Source handles the different implementations required.

DETAILED EXPLANATION

PLAYING AUDIO ON WEB

Playing audio for web is handled by the web page's audio element. Each Audio Source Component is responsible for creating a new audio element on the web page for it to run audio from.

After creating the audio element the Audio Source needs to setup, the element with the relevant audio clip (the source) and also make sure that the styling is set so that the audio element doesn't show it's default visuals.

PLAYING AUDIO ON MOBILE DEVICES

Playing audio on mobile devices is handled by Android native code (the Audio Source Component just tells the Android native code to play). The audio is handled by the iSound Java Class and mainly handles it in 2 steps. When the iSound instance (soundManager) is created the soundManager makes sure to load all of the sound effect clips so that they are optimized for reuse (background music is just loaded and played at the point the game want's it to start) and stores the optimized sounds in a map (using the sound clip file path as the key) so that the JavaScript side can easily tell the sound manager what clip is to play.

BOX COLLIDER

: SUBCLASSES FROM COMPONENT

CLASS DESCRIPTION

A Box Collider is a Component that acts as a container for all of the information required to define a box around the GameObject that it is attached to. These boxes are Axis-Aligned (so cannot be rotated) and may be offset from the GameObject's position.

The Box Collider Component is also responsible for checking if it is colliding with another Box Collider.

DETAILED EXPLANATION

DETECTING OVERLAPPING COLLIDERS

Firstly, the box collider needs to calculate the rectangles that both itself and the passed Box Collider encapsulate in the world (Rects). These Rects are calculated from the owning GameObject's position + the Box Collider's offset values and the width & height values that are set on the Box Colliders (the Box Colliders use their own width and height values not the width and height values of their owning GameObjects so that there can be differences between the size of the GameObject shown in through the renderer and the size of the collision range)

Then using both of the calculated Rects the Box Collider checks to see if any of the 4 corners of its own Rect is inside the Rect of the other Box Collider. If any of them are then it can be considered that a collision has occurred.

CAMERA CONTROLLER

: SUBCLASSES FROM COMPONENT

CLASS DESCRIPTION

A Camera Controller is a Component that is intended to be added to the scene camera. It enables the camera to follow another GameObject in the scene through smooth movement.

DETAILED EXPLANATION

FOLLOWING OTHER GAMEOBJECTS

The camera manages this smooth movement due to using standard interpolation (Lerping) equations to calculate its new position, specifically it uses its current global position as the Lerp's min value, the target GameObject's global position as the Lerp's max value and the deltaTime value as the Lerp point.

Since deltaTime is always a small value this means that the camera can never immediately teleport to its target location thus giving it the desired smooth movement.

MOVEMENT LIMITS

To make sure that the camera never moves too far from the main environment some limits have been placed on where the camera can move. Once the previously mentioned Lerp calculations have been handled the controller performs checks to see if the new position would be valid, if they are then the new position can be set on the parent GameObject otherwise the GameObject just stays where it is.

The limits for the X axis are that the camera's new X position must be between 500 and -500. This is so that the camera can show both sides of the environment without going too far off either side (which would show where the background sprites stop tiling)

The limit for the Y axis is just on the lower side of the world, the camera cannot have a Y position greater that 100 (this is due to positive values representing a downwards direction in the world because in screen coordinates 0,0 represents the top left corner and w,h represents the bottom right corner) this allows the camera to show the whole environment without worrying about how high target GameObjects may move upwards.

PHYSICS MOVEMENT

: SUBCLASSES FROM COMPONENT

CLASS DESCRIPTION

A Physics Movement is a Component that when added to a GameObject enables that GameObject to simulate as if it was being affected by physics. The Physics Movement Component also handles the resolution of collisions between Box Colliders

DETAILED EXPLANATION

CALCULATING SPEED AND POSITION

Calculating the equation of motion works due to the ability to find equations that solve for position, velocity and acceleration through differentiation and integration.

If you start from the equation to calculate an object's position, you can differentiate this to find the position of an object over time (its velocity) and then by differentiating again you can find the velocity of an object over time (its acceleration). This process can be reversed (although it is tricky), So if you have an equation for how the acceleration on an object is calculated (calculated from the forces acting on that object) then you can do the reverse (integration) to find the equations for calculating an objects speed and then integrate again to find the equation that calculate an object's position.

(To see the working for these equations, see Appendix 1)

When calculating the movement of an object the object's new speed is calculated first. The Physics Movement Component calculates the GameObject's new speed based off of the object's speed the frame before and takes into account the effect of gravity, air-resistance, and wind-speed (when calculating the speed values wind speed and air-resistance only affect the X axis and gravity and air-resistance affect the Y axis).

Once the new speed has been calculated the Physics Movement Component then calculates the new proposed position, this is not only calculated from the speed values, but gravity, air-resistance and wind speed also affect this (using the same limitations on axis)

COLLISION CHECKS

Before applying the new position, the Component makes sure to check if there would be any collisions occurring at that new position. To check against the scene colliders the Physics Movement Component is able to get a full list of all the colliders in the scene from the scene itself (due to a recursive function that gets all the colliders in a GameObject and all of its children).

The GameObject that contains the Physics Movement, its collider is responsible for checking if there are any collisions.

COLLISION RESOLUTION

From the previous collision checks the Physics Movement stores a list of all the colliders that it would collide with if it moved to the new position. From this it calculates the collider that is closest to the GameObject and uses this to resolve the collision.

Firstly, since all colliders are Axis Aligned Bounding Boxes the Component can determine what side of the colliding collider it is on (Up, Down, Left, or Right) and as such can easily reflect the object's speed values depending what side was hit (up or down the Y speed is multiplied by -1, right or left the X speed is multiplied by -1). A small adjustment to the X speed is needed due to the game having large windspeed values affecting objects. To prevent objects from getting stuck on walls the X speed must always be above 10 after a left or right collision (this causes the object to just gently slide down walls)

Then the Physics Movement handles any special collision events (defined by traditional JavaScript functions)

PLAYER CONTROLLER

: SUBCLASSES FROM COMPONENT

CLASS DESCRIPTION

A Player Controller is a Component that is responsible for both handling the aiming of the player's targeting cursor and the resolving the various actions that a player can do on each of their turns. It also handles the calculation of the player's health when damaged from various sources.

DETAILED EXPLANATION

AIMING

A GameObject is attached as a child of the main player gameobject, this gameobject has both Sprite Renderer and Button Controller attached to it to enable it to act as a responsive targeting reticle.

When the reticle button is held down it follows the direction of the mouse/ touch position (dependant on the platform that the game is playing on). The reticle does not directly follow the input position instead it gets the direction of the input position relative to the player GameObject, makes sure to normalize it and multiplies that direction by 50. This is used as the reticle's local position to ensure that it both orbits around the player GameObject and follows it during movement (the direction vector is normalized to ensure that the offset is always a constant amount away from the player no matter where the input is on the screen)

When the reticle's position changes this also updates another child of the player, the power meter GameObject. This power meter GameObject orbits the player in a similar manner to the reticle however its position is halfway between the player GameObject and the reticle GameObject. Also, it needs to update its attached Sprite Renderer's rotation value so that it is representative of the rotation around the player.

CHARGING

Both action buttons (the jump button & the fire Button) when held down sets it so that the current player's reticle GameObject is set to not be active (it will not render or update any of its components) and sets it so that the player's power meter GameObject is active. (The gameplay scene's Game Manager handles tracking which player is currently in control)

As the buttons are held down over time it increases a timer in the current player's controller (increases with delta time to ensure consistency across all devices/ platforms) and every quarter of a second the power value in the controller increases (this also resets the charge timer, and the power value cannot exceed 6). Also, when the power value increases the Sprite Renderer on the power meter GameObject has its Y offset increased so that the renderer shows an image that represents the increasing power on the controller.

JUMPING MOVEMENT

Although the movement of objects is handled by an attached Physics Movement Component this movement does not do much if left by itself (the player would just keep trying to fall and if stood on a collider would stay in place).

When the jump button is released it uses the targeting reticle GameObject's local position and normalizes it to get the direction that the player intends to move and multiplies it by the previously calculated power level and a scalar of 125. The player's Physics Movement component has its speed values set to the multiplied direction so that the player start's moving in the requested direction.

Finally, the controller tells the scene's game manager that the scene is now in the Player Jumping state so that it can determine when the turn is over (when the player stops moving)

FIRING PROJECTILES

This works the same as how a player's jump is handled except instead of affecting the player's Physics Movement Component, it instead spawns a new projectile prefab. This new projectile has its Physics Movement Component affected in the same way as the frog jumping (direction from targeting reticle multiplied by the current power level) also the physics movement component is told to add the player character to its collision layers so that the projectile cannot collide with the player that spawned it.

Again, the scene's game manager is told to change state however this time it is told to be in the Player Fired state as the turn over condition is handled by the projectile instead (when the projectile either collides with something or falls too far down in the world)

PLAYER HEALTH

All player GameObjects in the scene handle when their health is being reduced (triggered by either a projectile colliding with the player or the player falling too far down the world and out of bounds). The main check that happens here is if the player's health value falls below is equal to 0. If this happens then the player updates the global "WinningPlayer" property with the opposite player's number (either 1 or 2), it then gets the game to move over to the Game Over Scene.

PROJECTILE CONTROLLER

: SUBCLASSES FROM COMPONENT

CLASS DESCRIPTION

A Projectile Controller is a component that is responsible for handling the animation that a projectile has to show its different sprites based on its movement speed and for calculating the sprite rotation based on the projectile's movement direction.

The Projectile Controller also handles despawning the projectile it is attached to if the projectile has moved to far from the main environment.

DETAILED EXPLANATION

ANIMATION

For this animation to work properly the GameObject that the Projectile Controller is attached to also needs to have a Physics Movement Component and a Sprite Renderer Component attached to itself.

To determine how the Projectile Controller should affect the GameObject's Sprite Renderer it relies on the speed values calculated by the Physics Movement Component. Specifically, it uses the speed's magnitude (from both axis) to determine what image from the Sprite Renderer's sprite sheet to use (handled by changing the Sprite Renderer's Y Offset).

When calculating the Sprite's rotation, the controller again uses both the X and Y speed of the Physics Movement Component however instead of working from the speed's magnitude the controller takes both values as a direction vector (both values are divided by the previously calculated magnitude so that the direction vector is a normalized vector), This direction vector is compared to a standard Up vector (0, 1) to get the angle between them.

HANDLING OUT OF BOUNDS

As part of its update loop, the Projectile Controller checks to see if the projectile it is attached to has moved to far from the main gameplay environment. It handles this by doing a single simple check, If the projectile has moved too far down the screen (specifically if its y position is too great due to the screen coordinate being (0, 0) on the top left and (x, y) on the bottom right) then the projectile controller can let the current scene know that the object it is attached to can be destroyed.

The controller only needs to check the Y-Axis since the projectiles move with physics. Because of this the controller can assume that the projectile will always have some amount of force dragging it to the ground (all objects that move with physics have gravity affecting them on the Y-Axis)

Whilst this aspect of the Projectile controller does not require a Physics Movement Component to be attached to the GameObject, as previously mentioned having one attached ensures that the projectile will have a distinct end point to its lifetime.

SPRITE RENDERER

: SUBCLASSES FROM COMPONENT

CLASS DESCRIPTION

A Sprite Renderer is a Component that is responsible for rendering Images to the screen, these Images exist in the game's World Space and so the Sprite Renderer is also responsible for calculating the Sprites position and size in screen space coordinates

This Sprite renderer component is unique against other render components in that it also enables the World Space Images to be rotated and flipped.

DETAILED EXPLANATION

RENDERING IMAGES

TEXT RENDERER

: SUBCLASSES FROM COMPONENT

CLASS DESCRIPTION

A Text Renderer is a Component that is responsible for showing text-based UI to the screen and scaling the screen position and text size so that it is consistent across platforms/ devices.

DETAILED EXPLANATION

RENDERING TEXT

When rendering sprites there is a very specific set of actions that must happen for a sprite to render so that it can take into account complex transformations such as rotation and mirroring.

Firstly, the renderer needs to transform the canvas that the game is drawn on so that if the sprite is to be drawn flipped it will still be drawn in the same position on the screen, it manages this by moving the canvas (by its width and/or height if the sprite is to be flipped on the X and/ or Y axis) and then scales the canvas by -1 on the same axis' as the movement. Then the renderer transforms the canvas again but this time it moves the canvas by the sprites screen position then rotates it and then moves it back.

After the complex transformations have been handled the sprite can be drawn to the canvas. Before rendering the renderer needs to calculate the sprite's screen position relative to the scene's camera so that the sprite's world position is converted to screen position. The sprite is also scaled by the camera's scale value (both the sprite's size and screen position values are scaled) so that the sprites render at different sizes/ screen positions depending on the device/ platform that they're on, this allows the game to scale the game so that It is consistent across those devices/ platforms.

Finally, it is important to reverse the previous transformations so that the canvas is back to its default state for whatever GameObject gets rendered next (the transformations must be done in reverse order).

UI RENDERER

: SUBCLASSES FROM COMPONENT

CLASS DESCRIPTION

The UI Renderer is a Component that is responsible for rendering images to the screen.

DETAILED EXPLANATION

RENDERING IMAGES

Unlike the Sprite Renderer Component, the UI Renderer does not need to transform between world and screen space as the images that it renderers are to be shown as the Games UI and as such exist in screen space already, Like the Sprite Renderer, the UI Renderer also needs to scale the screen position and image size so that it is consistent across platforms/ devices.

REFERENCES

RESEARCH -

JavaScript - https://www.javascript.com/

Mozilla JavaScript Documentation - https://developer.mozilla.org/en-US/docs/Web/JavaScript

W3 Schools JavaScript Tutorials - https://www.w3schools.com/js/default.asp

HTML5 Game Engines - http://html5gameengine.com/

Phaser - https://phaser.io/

WebGL - https://www.khronos.org/webgl/

ASSETS

SOUNDS

Tired traveller on the way to home – Andrew Codeman - https://freemusicarchive.org/music/Andrew_Codeman/Fall_Trajectory/Andrew_Codeman_Fall_Trajectory_01_Tired_traveler_on_the_way_to_home

Pop Sound Effect - https://freesound.org/people/greenvwbeetle/sounds/244656/

FONTS

Forest Thing Font - https://www.1001freefonts.com/forest-thing.font

All other assets in the game have been created by myself.

APPENDIX A – EQUATIONS OF MOTION WORKING

$$m \begin{bmatrix} \ddot{x} \\ \ddot{y} \end{bmatrix} = \begin{bmatrix} 0 \\ -mg \end{bmatrix} + \begin{bmatrix} -\lambda \dot{x} \\ -\lambda \dot{y} \end{bmatrix} + \begin{bmatrix} A \\ 0 \end{bmatrix}$$

$$m\ddot{x} = -\lambda \dot{x} + A$$

$$\ddot{x} = \frac{-\lambda}{m} \dot{x} + \frac{A}{m}$$

$$m\ddot{y} = -mg - \lambda \dot{y}$$

$$\ddot{y} = -g - \frac{\lambda}{m} \dot{y}$$

X Direction Calculations

$$\ddot{x} + \frac{\lambda}{m}\dot{x} = \frac{A}{m}$$

// Non Homogenous Equation

Homogenous Solution

$$\ddot{x} + \frac{\lambda}{m}\dot{x} = 0$$

$$a^2 + \frac{\lambda}{m}a = 0$$

$$a\left(a + \frac{\lambda}{m}\right) = 0$$

$$a_1 = 0 \qquad a_2 = \frac{-\lambda}{m}$$

$$x_H = K_1 e^{0t} + K_2 e^{\frac{-\lambda}{m}t}$$

$$= K_1 + K_2 e^{\frac{-\lambda}{m}t}$$

Particular Solution

$$x_p = \alpha t$$

$$\dot{x}_p = \alpha$$

$$\ddot{x}_p = 0$$

$$(1 \times 0) + \left(\frac{\lambda}{m} \times \alpha\right) + (0 \times \alpha t) = \frac{A}{m}$$

$$\frac{\lambda}{m}\alpha = \frac{A}{m}$$

$$\lambda\alpha = A$$

$$\alpha = \frac{A}{\lambda}$$

$$x_p = \frac{At}{\lambda}$$

Associated Solution

$$x(t) = x_H + x_p$$

$$= K_1 + K_2 e^{\frac{-\lambda}{m} t} + \frac{At}{\lambda}$$

$$\dot{x}(t) = \frac{-\lambda}{m} K_2 e^{\frac{-\lambda}{m} t} + \frac{A}{\lambda}$$

Solve for K_1 & K_2

$$x(0) = posX \quad , \quad \dot{x}(0) = velX$$

$$\dot{x}(0) = \frac{-\lambda}{m} K_2 e^{\frac{-\lambda}{m} \times 0} + \frac{A}{\lambda} = VelX$$

$$\frac{-\lambda}{m} K_2 + \frac{A}{\lambda} = VelX$$

$$\frac{-\lambda}{m} K_2 = VelX - \frac{A}{\lambda}$$

$$-\lambda K_2 = m\left(velX - \frac{A}{\lambda}\right)$$

$$K_2 = \frac{mvelX - \frac{Am}{\lambda}}{-\lambda}$$

$$x(0) = K_1 + K_2 e^{\frac{-\lambda}{m} \times 0} + \frac{A \times 0}{\lambda} = posX$$

$$= K_1 + K_2 = posX$$

$$K_1 = posX - K_2$$

$$= posX - \frac{mvelX - \frac{Am}{\lambda}}{-\lambda}$$

Movement Equation in the X-Axis

$$x(t) = \left(posX - \frac{mvelX - \frac{Am}{\lambda}}{-\lambda}\right)$$

$$+ \left(\frac{mvelX - \frac{Am}{\lambda}}{-\lambda}\right) e^{\frac{-\lambda}{m} t}$$

$$+ \left(\frac{At}{\lambda}\right)$$

$$\dot{x}(t) = \left(\frac{-\lambda}{m}\right) \times \left(\frac{mvelx - \frac{Am}{\lambda}}{-\lambda}\right) e^{\frac{-\lambda}{m} t}$$

$$+ \left(\frac{A}{\lambda}\right)$$

Y Direction Calculations

$$\ddot{y} + \frac{\lambda}{m}\dot{y} = -g$$ // Non-Homogenous Equation

Homogenous Solution

$$\ddot{y} + \frac{\lambda}{m}\dot{y} = 0$$

$$b^2 + \frac{\lambda}{m} b = 0$$

$$b\left(b + \frac{\lambda}{m}\right) = 0$$

$$b_1 = 0 \qquad b_2 = \frac{-\lambda}{m}$$

$$y_H = K_3 e^{0t} + K_4 e^{\frac{-\lambda}{m}t}$$

$$= K_3 + K_4 e^{\frac{-\lambda}{m}t}$$

Particular Solution

$$y_p = \alpha t \qquad (1 \times 0) + \left(\frac{\lambda}{m} \times \alpha\right) + (0 \times \alpha t) = -g$$

$$\dot{y}_p = \alpha \qquad \frac{\lambda}{m}\alpha = -g$$

$$\ddot{y}_p = 0 \qquad \lambda\alpha = -gm$$

$$\alpha = \frac{-gm}{\lambda}$$

$$y_p = \frac{-gmt}{\lambda}$$

$$y(0) = K_3 + K_4 e^{\frac{-\lambda}{m} \times 0} - \frac{gm \times 0}{\lambda} = P$$

$$K_3 + K_4 = posY$$

$$K_3 + \left(\frac{mvelY + \frac{gm^2}{\lambda}}{-\lambda} \right) = posY$$

$$K_3 = posY - \left(\frac{mvelY + \frac{gm^2}{\lambda}}{-\lambda} \right)$$

Movement Equations in the Y-Axis

$$y(t) = \left(posY - \left(\frac{mvelY + \frac{gm^2}{\lambda}}{-\lambda} \right) \right) + \left(\frac{mvelY + \frac{gm^2}{\lambda}}{-\lambda} \right) e^{\frac{-\lambda}{m} t} - \left(\frac{gmt}{\lambda} \right)$$

$$\dot{y}(t) = \left(\frac{-\lambda}{m} \right) \times \left(\frac{mvelY + \frac{gm^2}{\lambda}}{-\lambda} \right) e^{\frac{-\lambda}{m} t} - \left(\frac{gm}{\lambda} \right)$$

Associated Solution

$$y(t) = y_H + y_p$$

$$= K_3 + K_4 e^{\frac{-\lambda}{m} t} - \frac{gmt}{\lambda}$$

$$\dot{y}(t) = \frac{-\lambda}{m} K_4 e^{\frac{-\lambda}{m} t} - \frac{gm}{\lambda}$$

Solve for K_3 & K_4

$y(0) = posY$, $\dot{y}(0) = velY$

$$\dot{y}(0) = \frac{-\lambda}{m} K_4 e^{\frac{-\lambda}{m} \times 0} - \frac{gm}{\lambda} = velY$$

$$\frac{-\lambda}{m} K_4 - \frac{gm}{\lambda} = velY$$

$$\frac{-\lambda}{m} K_4 = velY + \frac{gm}{\lambda}$$

$$-\lambda K_4 = mvelY + \frac{gm^2}{\lambda}$$

$$K_4 = \frac{mvelY + \frac{gm^2}{\lambda}}{-\lambda}$$