# THE PSYCHTOOLBOX GAME ENGINE

**August 2, 2019**

Nigel Bess

# Contents

## 1 Overview

The purpose of the game engine is to simplify software development, troubleshooting, and to remove the developer's need to interact with underlying game structure. It does this by encapsulating tasks within the software into objects of type `GameObject`, and interacting with `GameObjects` automatically.

Upon instantiation of a game engine object, it finds all the `GameObjects` in the workspace. When the game engine is started it first calls the `Awake()` method on each of these `GameObjects`. Then it enters a loop in which it calls the `Update()` method on each `GameObject`, and continues to do so until the `Quit()` method is called on the game engine.

This game engine is used for the Autobehavior Training Rig software, but could be used to create any 2D game.

## 2 Setting up and Running a Game

1. Create all `GameObjects` desired.
2. Initialize desired parameters of all `GameObjects`.
3. Create a `GameEngine`.
4. Start the game

## 3 Detailed Documentation of Data Types

This section includes detailed usage of the each basic type in the game engine. Any methods or properties not listed are only used for the underlying game engine structure.

### 3.1 GameEngine

There should exist one and only one `GameEngine` in the workspace. Simply speaking, the `GameEngine` finds all `GameObjects`, calls each `GameObject's Awake()` method, then calls each `GameObject`'s `Update()` method repeatedly until the game is quit.

### 3.1.1 Constructor: GameEngine(sceneFileName)

Finds all objects in the workspace. If using the optional parameter, [sceneFileName], it loads the workspace saved in [sceneFileName].mat.

### 3.1.2 GameEngine.Start()

The `Start` method begins the game. First, it passes a reference to itself to each `GameObject` in the workspace. It then calls the `Awake` method on each `GameObject`. It then initializes the Renderer. Finally, it enters the main game loop, in which the `Update` method is called for each enabled `GameObject`. **Note:** The order in which `Update` is called for each `GameObject` is consistent, but for best practice code should be written in a way that order does not matter.

### 3.1.3 GameEngine.Quit()

The `Quit` method breaks from the main game loop. It calls the `OnQuit` method for each `GameObject` if available. It then closes the rendering window.

### 3.1.4 GameEngine.FindObjectsOfType(typeChar)

Inputs `typeChar`, a character array defining the name of a class or base class (case sensitive). Returns all objects (from those in the workspace before the game engine was created) of class `typeChar`. This includes objects that inherit from class `typeChar`.

### 3.1.5 GameEngine.minTimeDelta

Private property defining a time (in seconds) to pause between frames. Set to zero for maximum performance without background processes. Increase the value to improve background process performance on a CPU with low thread count. Default value 0.01 seconds. Public setter at `GameEngine.SetMinTimeDelta(time)`.

### 3.1.6 GameEngine.GetTime()

Returns time (in seconds) since the game was started.

### 3.1.7 GameEngine.GetTimeDelta()

Returns time (in seconds) since the last frame.

## 3.2 GameObject

`GameObject` is the class from which all objects interacting with the game loop should inherit.

### 3.2.1 GameObject.Game

The current active `GameEngine`.

### 3.2.2 GameObject.Renderer

The current active `Renderer`.

### 3.2.3 GameObject.Awake()

Called before the first frame, and before rendering has begun.

### 3.2.4 GameObject.enabled

A public property (boolean) determining if the object should be updated this frame.

### 3.2.5 GameObject.Update()

Called once per frame, as long as the object is enabled.

### 3.2.6 GameObject.DelayedCall(methodName,time,param0,param1...paramN)

Calls [methodName]([params]) on this `GameObject` after [time] seconds. methodName should be a character array.

### 3.2.7 GameObject.DisableFor(time)

Prevents the `Update` method from being called for [time] seconds;

### 3.2.8 GameObject.StopAllDelayedCalls()

Prevents any pending delayed calls from resolving. If the object was temporarily disabled, it will not automatically re-enable. This also stops any delayed calls that were still going to resolve this frame.

### 3.2.9 GameObject.OnQuit()

Called when the game is quit.

### 3.2.10 GameObject.OnError()

Called when the game encounters a fatal error. If an error is encountered in a `GameObject.OnError`, the game engine will throw a warning.

### 3.2.11 GameObject.BaseUpdate() (Use sparingly)

`BaseUpdate` works exactly the same as the Update method, and while it could be used interchangeably, in shouldn't be. The purpose of `BaseUpdate` is to pass an update function from a base class to its children without impeding on the children's update functions.

## 3.3 Renderer

There should exist no more than one `Renderer` in the workspace. The `Renderer` is a `GameObject` that handles all rendering of images to the screen. When the `Renderer`'s `Update()` function is called, it draws each instance of each `Renderable`'s image to the screen, according to the `Renderable`'s render order.

### 3.3.1 Constructor: Renderer(screenNum,backgroundColor,percentRect)

Initializes a `Renderer` to output to the screen defined by [screenNum], with a default background color of [backgroundColor]. [percentRect] defines the fraction of the screen to be rendered to, in the form [xMin, yMin, xMax, yMax]. By default, [percentRect] is [0, 0, 1, 1], resulting in rendering to the entirety of the screen. screenNum: 0 - single display mode, 1 - main display, 2- secondary display.

### 3.3.2 Renderer.WindowSize()

Returns the size of the rendering window (in pixels) in the form: [width,height].

### 3.3.3 Renderer.SetBackgroundColor(color)

Sets the background color to [color].

#### 3.3.4 Renderer.ResetBackgroundColor()

Resets the background color to the default color.

## 3.4 Renderable

`Renderable`s are `GameObject`s that render to the screen. Each instance of a renderable has a single image, which can be simultaneously rendered at one or more places on the screen. By default a renderable will only have one image instance, but more can be added by calling `Renderable.InstantiateNew()`.

#### 3.4.1 Renderable.GenerateImage()

Defines the `Renderable`'s image (in image matrix form), which gets rendered to the screen. It must be defined for any `Renderable` object.

#### 3.4.2 Renderable.PngToImage(pngFileName)

Finds PNG file named [pngFileName].PNG. Returns an image matrix corresponding to this image file.

#### 3.4.3 Renderable.rootPosition

A protected property defining the root position of this `Renderable` (in pixels with the form [x,y], relative to the global root position of the `Renderable`'s parent). It has a public setter at `Renderable.SetRootPosition`.

**Note 1:** This is not necessarily where the image(s) will be rendered. Each image instance has its own position, which is defined relative to the root position. For example, lets say a `Renderable` has a root position of [1, 2]. Its has two image instances at positions [1, 1] and [100, 100]. The images would be rendered at [2, 3] and [101, 102], respectively.

**Note 2:** The above note assumes that the `Renderable` in question has no parent. `Renderable.rootPosition` is defined relative to the global root position of the parent.

### 3.4.4 Renderable.position

A protected property defining the relative positions (in pixels) of all image instances of this `Renderable`. Each row contains the position of its respective instance in the form [x,y]. [0,0] will render at the `Renderable`'s global root position. It can be modified by the function `Renderable.SetInstancePosition([New position], [Instance number])`.

### 3.4.5 Renderable.size

A protected property defining the size (in pixels) of each image instance. Each row has the form [width, height] for its respective instance.

### 3.4.6 Renderable.InstantiateNew(position,size)

Creates a new image instance for this `Renderable`. The instance will be located at relative position defined by [position] as [x, y] (in pixels). Its size will be defined by [size] as [width, height] (also in pixels). If these parameters are omitted, the default position will be [0,0] and the default size will match the size of the first instance.

### 3.4.7 Renderable.Remove(instance)

Removes image instance defined by integer number [instance].

### 3.4.8 Renderable.screenBounded

A protected property (boolean) determining whether the `Renderable` is allowed to leave the screen. If `screenBounded` is true, each image instance of the `Renderable` will have its position clamped such that its entire rect is always visible on screen.

For `Renderables` with multiple image instances, `Renderable.screenBounded` will modify each instance's position so that it remains on the screen. For `Renderables` with a single instance, the `Renderable`'s root position will be modified.

### 3.4.9 Renderable.renderLayer

A public property (double) determining the order in which the `Renderable` is rendered to the screen. A `Renderer` with **larger** number at `renderLayer` will render **below** other `Renderables` on screen.

This property was intentionally left public (in violation of the convention described in 4.2) in order to simplify the sorting of the renderables by render order.

### 3.4.10 Renderable.RenderAfter(other)

Causes this `Renderable` to be rendered behind `Renderable` [other]. It does this by setting `renderLayer` to [other].`renderLayer`.

### 3.4.11 Renderable.GetScreenHits(index, instance)

Returns whether any part of the image instance defined by integer number [instance] is outside of the rendering window along specified index.
Index: 1 - Left/Right, 2 - Up/Down
Returns: -1 - Left/Top, 0 Inside window, 1 - Right/Bottom

### 3.4.12 Renderable.Distance(other, index)

Returns the distance (in pixels) between the global root position of this `Renderable` and the global root position of `Renderable` other. Index parameter is optional, and returns distance along a specified axis.
Index: 1 - Horizontal, 2 - Vertical

## 3.5 Parenting of Renderables

Say you want to have two or more renderables move in unison, or define relative motion between renderables. This is accomplished using parenting.

One renderable can be defined as the parent. The other renderable would be the child. The child's root position is defined relative to the global root position of the parent. Assuming that the parent has no parents of its own, its global root position, would simply be its root position.

Each renderable can have up to one parent, and any number of children. The height of a tree is limited by MATLAB's maximum recursion limit of 500.

### 3.5.1 Renderable.SetParent(other)

Sets the parent of the renderable to [other] (which should also be a renderable).

### 3.6 PhysicsObject

`PhysicsObjects` are `Renderables` that maintain persistent velocity and update their root position automatically.

#### 3.6.1 PhysicsObject.Velocity

Protected property defining the 2D velocity (in pixels per second) with the form [xVelocity,yVelocity]

#### 3.6.2 PhysicsObject.SetVelocity(velocity,index)

Setter for `PhysicsObject.velocity`. Optional parameter [index] is used to alter only one entry of `PhysicsObject.velocity`

## 4 Conventions

### 4.1 Case Conventions

For reference, Pascal case and Camel case are defined as follows:

```
PascalCase
camelCase
```

| | |
|---|---|
| Class Names | Pascal |
| Method Names | Pascal |
| Private/Protected Property Names | Camel |
| Public Property Names | Camel |
| Constant Property Names | Camel |
| Property Names Referring to Singletons | Pascal |

### 4.2 Access

Methods should be used to interact between objects, and only in very rare situations should public properties be used. Almost every property in the game engine is either private, protected or constant, and it is recommended to follow this convention when creating games.

The exception was made for `GameObject.enabled` in order to match the Unity Game Engine.

## 4.3 Other

It is recommended to follow the S.O.L.I.D. object-oriented programming principles when possible.