The   
Flipper Framework

Code Style Guide

Table of Contents

[File Structure 3](#_Toc166531494)

[Hierarchy 3](#_Toc166531495)

[File Contents 4](#_Toc166531496)

[Code Style Guide 6](#_Toc166531497)

[Naming 6](#_Toc166531498)

[Keywords 7](#_Toc166531499)

[Variables – By purpose 8](#_Toc166531500)

[Variables – By data type 9](#_Toc166531501)

[Headers – 11](#_Toc166531502)

[Assets – 13](#_Toc166531503)

[Formatting 14](#_Toc166531504)

[Keywords - 14](#_Toc166531505)

[Indentation – 15](#_Toc166531506)

[Spacing - 15](#_Toc166531507)

[Conventions 17](#_Toc166531508)

[Calculating Velocity 17](#_Toc166531509)

[Setting important values 18](#_Toc166531510)

[Action Rules 18](#_Toc166531511)

[Comments Rules 19](#_Toc166531512)

[Class Structure 20](#_Toc166531513)

# File Structure

## Hierarchy

Below is a graph showing the hierarchy of every major folder. When going through the source files, this is how they are organised.

A diagram of a game

Description automatically generated

## File Contents

Below is a table of every file in the above, detailing what they contain.

|  |  |
| --- | --- |
| Folder | Contents & Purpose |
| !Framework | This contains all files directly related to the Flipper Framework that the game should be built from. |
| Game | This contains content assets directly related to a game developers intend to make. This includes levels, new assets, or anything unique to the game developers intend to make and not the framework they’re using. |
| Unity | This contains any assets not created for the framework or game, and instead, acquired externally, such as through the Unity Asset Store. |
| Audio | Any files that handle or produce audio. Organised by purpose and sort of thing they’re producing sounds for. |
| Building Blocks | This contains all the GameObjects that can be placed in a scene to make the game. The most important folder that acts like a level editor. Users should be able to go into this folder and build a level from its contents. |
| Scripts | All C# scripts that decide how the game works. These are split across what features they perform, or what sort of elements they’d be attached to as components. |
| Visual | This refers to any asset the player will see. Whether it be an object, material, sprite, or anything that handles the above. |
| Common (Audio) | Any sound effects that will be used a lot, across a variety of objects. |
| Characters (Building Blocks) | Any prefabs or scriptable objects that make up the characters the player will play as. |
| Core Objects (Building Blocks) | The most used objects used repeatedly in Sonic levels. |
| Dangers (Building Blocks) | Any prefab places with the intent to harm or endanger the player. |
| Extra Objects (Building Blocks) | Any prefabs that are more situational, but aid the player. |
| Pathing (Building Blocks) | Any prefabs that make up rails, or automatic paths. Usually based on splines. |
| Platforms (Building Blocks) | Prefabs for objects the player can stand on, but will have effects like moving. |
| Stage Models (Building Blocks) | Object files rather than prefabs, these are geometry the user can build levels out of. |
| Core Game Functionality (Scripts) | Scripts that are required for the game to progress optimally, and will affect the whole level, rather than specific objects. |
| Effects (Scripts) | Scripts that control visual effects. |
| Enemies (Scripts) | Scripts that control the behaviour and control of enemies. In other words, the AI. |
| General Use (Scripts) | Scripts that can be used or referenced repeatedly. This includes storage of data types like enums, custom inspectors, and general-purpose components that can be applied to most objects. |
| Objects (Scripts) | Any script that turns a regular object into a specific interactable object. |
| Other (Scripts) | Any script that doesn’t fit into the other categories. |
| Player (Scripts) | Scripts applied in the player object prefabs. These include the actions characters can perform, how they interact with objects, how they control, and more. |
| UI (Scripts) | Scripts that control interaction and effects of interfaces. |
| Character Assets (Visual) | Objects, materials and textures, organised by what character they belong to. |
| Effects (Visual) | Any assets related to VFX, including objects, prefabs and materials. |
| General Materials (Visual) | Materials that can be applied to a variety of objects and geometry, and their textures. |
| Rendering (Visual) | Assets related to the Universal Render Pipeline and lighting. |
| Shaders (Visual) | Custom shaders that allow all materials to work. |
| Terrain (Visual) | The terrain assets and brushes that can make them. |
| UI (Visual) | Any art assets used in the interfaces, like fonts, sprites and backgrounds. |
| Unique Objects (Visual) | Further organised into folders for specific objects (ordered like how they are in building blocks). This includes the materials, objects and textures used for specific objects. |

# Code

## Naming

When writing code, it is incredibly important to use consistent names that inform future readers what that element is used for, and what type of element it is.

What follows is the naming convention used for each element individually. Please note that many elements, especially variables, will use multiple of these (E.G. a public global Boolean uses the naming conventions of all three).

Naming conventions will usually be applied for one of the following reasons, all related to readability.

* To make a variable’s origins obvious at a glance.
* To make a variable’s purpose obvious at a glance.
* To make searching for elements easier using CTRL + F.
* To make the code closer to a human language.

### Keywords

For simplicity’s sake, frequently repeated conventions or terminology will be described here, rather than when used.

|  |  |  |
| --- | --- | --- |
| Keyword / Convention | Description | Example(s) |
| Prefix | A word, number, symbol, or letter placed before the rest of the name. | iCounter  isRunning |
| Suffix | A word, number, symbol, or letter placed after the rest of the name. | counter\_  SetSpeed1 |
| Symbols | Symbols can be dangerous in C# coding because they typically have specific functions. As such, only specific ones can be used in names, especially as suffixes or prefixes. | Underscores can be used.  Exclamation marks and full stops cannot. |
| File Type | Every asset in Unity has a purpose and fits into a type befitting this. These are separate from the actual file format. The type is usually included as a prefix in naming files. | Script, *a C# script.*  Texture, *a 2D art image like a PNG or JPEG.* |
| Data Type | The type of data the variable handles. | String  Integers |
| Camel Case | Writing without spaces or punctuation, separating words via uppercase, but starting with lowercase. | exampleHealth  playerMovement |
| Pascal Case | Camel case where the first letter is upper case. | ExampleHealth  PlayerMovement |
| Snake Case | Separating words with an underscore. | example\_health |
| Hungarian Notation | Prefixed with the data type. This is not commonly used in C#. | fExampleHealth  iCounter |

### Variables – By purpose

|  |  |  |  |
| --- | --- | --- | --- |
| Purpose | Description | Convention | Example |
| Global / class fields | Declared in a class but outside of the methods. Therefore, can be accessed by all of them. | Prefix with an underscore.  Always declare with access modifier. | Public float \_currentHealth  Private Vector3 \_currentOffset |
| Local & parameters | Declared during or when setting a function/method. Not stored once the function has been run. | No unique convention. | counter |
| Serialized Stat | A global serialized variable can be set per class instance (i.e. in the Unity editor).  If that variable is intended to be easily changed to vary the instance, and is not a reference to another asset, it is a stat.  This includes variables set by scriptable objects. These typically are not changed during runtime. | Prefix & Suffix with an underscore. | \_maxHealth\_  \_startAcceleration\_  \_startJumpHeight\_ |
| Static | A single instance of this variable is shared among all object types. | Prefix with “s”, in addition to the global prefix. | s\_Instance |
| Counter | A temporary variable only used for keeping track of passes in a loop. | Can be one letter, but must indicate the data type. | int i  string str  GameObject GO |
| Variables around one subject | A series of variables all involving the same feature and by extent, share a word. | Must be declared together, line by line. The first must use the subject’s full name, but the following can use shortenings. | acceleration accellDecay maxAccel |

### Variables – By data type

|  |  |  |  |
| --- | --- | --- | --- |
| Purpose | Description | Convention | Example |
| All |  | Unless otherwise stated all variables should; Be nouns.  Be singular. Be full names, not shortenings.  Use Camel case. | Private float acceleration.  Public int jumpCount |
| Class or Unity-specific | A variable referencing an element unique to Unity. I.E. an asset, component, or another class. | Pascal case | RigidBody  SoundManager |
| Booleans | True or false | Prefix with a word that implies the name to be a question. Can be a verb. | isRunning  canRun  isInvincible |
| Curves | Data types that store data in graphs with an x and y value. Typically used to get a y value from the x. | Pascal case. Written as “y By x”. The vertical scale that a value of is gained when the horizontal scale is given in. | DamageByCurrent Health |
| Enums  (Type) | A group of selectable constants, represented as names.  These must be created as custom data types. | Camel Case. Plural. If cannot be easily pluralised, suffix with “Types”. If not defined in a global script that only exists to store Enums, prefix with “Enum”. | EnumPlayerStates  EnumAttackTypes |
| Enums  (Instance) | A variable using a custom enum data type that stores one of the constants. | Prefix with “what”, to imply it can be one of multiple options. | whatState  whatType |
| Lists & Arrays | A variable that holds multiple variables of the same type in an order. | Prefix with “listOf” (camel or pascal depending on contents). | listOfBounceHeights  ListOfStoredEnemies |
| Structs  (Type) | A group of unlinked fields that can be different data types. Can also include methods but should only be used for data.  These must be created as custom data types. | CamelCase Plural. If not defined in a global script that only exists to store structs, prefix with “Struct”. | StructJumpStats  OutputUIs |
| Structs  (Instance) | A variable using a custom struct data type storing data instances. | Plural | jumpStats |
| Relevant Objects in Calculations | If a variable is storing a specific known GameObject that shares a similar name to an attached script it should have. | Suffix with “Object”. | SpringObject  DashPadObject |
| Relevant external Scripts in calculations | If a variable is storing a specific type of script that should be attached to an object with a similar name. | Suffix with “Script”.  This will make it easier to know when calling the data and methods of the object, or the object itself. | SpringScript  DashPadScript |
| Delegates  (Type) | Represents a method with a particular parameter list and return type. Must be instantiated as a custom data type first. | Pascal Case.  Prefix with “Delegate” to turn into a verb. | DelegateTurning  DelegateDamage |
| Delegates  (Instance) | A variable that stores references to methods matching its type. | Pascal Case.  Prefix with “Call” to turn into a verb. | CallAcceleration  CallDamage |
| Events | A multicast delegate that can notify several methods at once. Unity provides Unity Events alongside built-in C# events. | Pascal Case.  Prefix with “On”. | OnGrounded  OnDamaged |

### Headers –

This refers to the names given above any block of code, such as classes or methods. The following table will describe the types of editors, and how they can be adjusted based on purpose or data.

|  |  |  |
| --- | --- | --- |
| Header | Convention | Example |
| All | Use Pascal casing |  |
| Class | Noun, plural unless otherwise stated. | PlayerPhysics |
| Class inherited from MonoBehaviour | Prefix with “S\_” for “script”. This is because the class name defines the file name. See more in Asset Naming | S\_PlayerPhysics |
| Class inherited from Scriptable Object | Prefix with “S\_O\_”  Separated by underscores so the file still comes up when searching for all lists via “S\_” | S\_O\_CharacterStats  S\_O\_CameraStats |
| Static Class | Prefix with “S\_S\_” | S\_S\_GameManager |
| Script handling a performable action/character state. | Prefix with “S\_Action\_” followed by the action. | S\_Action\_Jump  S\_Action\_RailGrind  S\_Action\_HomingAttack |
| Script handling a sub action. | Prefix with “S\_SubAction\_” followed by the additional action the character can perform when still performing a main action. | S\_SubAction\_Roll  S\_SubAction\_Skid  S\_SunAction\_Boost |
| Script handling the behaviour of an enemy. | Prefix with “S\_AI\_” followed by the enemy or type of enemy if applicable to multiple. | S\_AI\_RailEnemy  S\_AI\_MotoBug |
| Script running in the background supporting other player scripts. | Prefix with “S\_Handler\_” followed by the feature it supports.  If it supports an action, give the action name. | S\_Handler\_HomingAttack  S\_Handler\_RingRoad  S\_Handler\_Health |
| Script handling effects when interacting with specific external objects in a level. | Prefix with “S\_Interactions\_” followed by the type of objects or type of interactions it handles. | S\_Interactions\_Pathers  S\_Interactions\_Attacks  S\_Interactions\_Objects |
| Simple script that stores data and some behaviour for an object. | Prefix with “S\_Data\_” followed by the object anything with this component should be. | S\_Data\_Spring  S\_Data\_Checkpoint  S\_Data\_DashPad |
| Script that affects an object, either in game or in editor. | Prefix with “S\_Control”, followed by the object anything with this component should be. | S\_Control\_MovingPlatforn  S\_Control\_Zipline |
| Script that performs methods when a player comes into contact with its object. | Prefix with “S\_Trigger” followed by the type of object it will affect. | S\_Trigger\_Camera  S\_Trigger\_CinemaCamera  S\_Trigger\_RailEnemy |
| Script handling the interactions or spawning of a UI Object. | Prefix with “S\_UI” followed by the theme of the object (or group of objects) it is responsible for. | S\_UI\_Boost  S\_UI\_PauseMenu |
|  |  |  |
| Interfaces | Singular noun. Prefix with “I” | IAction |
| Methods | Phrase as a command/verb. If necessary, use a prefix. | ApplyGravity  AssignStats |
| Method returning a Boolean | Phrase as a yes or no question. | isGrounded |
| Method mainly responsible for changing values of a major variable, along with things related to it. | Prefix with “Set”, followed by the variable’s name. | SetIsGrounded  SetCoreVelocity  SetLateralSpeed |
| Method that is only called by events. | Prefix with “Event”, followed by a name like the event that should call it. | EventOnGrounded  EventOnLoseGround |

### Assets –

While not involved in the code, this section shall cover the naming conventions followed for assets according to their File Type. Anything contained in the source files should follow at least one of the following conventions. This technique is more common in Unreal Engine projects but can be followed anywhere. This will not cover all file types, as some are rarely used so do not need assistance in being easier to search for and select.

|  |  |  |
| --- | --- | --- |
| File Type | Convention | Example |
| Script | Prefix with “S\_” | S\_PlayerPhysics |
| Any 2D image. JPEG, PNG. | Prefix with “Tex\_” | Tex\_4x4Grid |
| An image intended to be used as a normal map. | Suffix with “\_Normal” | Tex\_Bumper\_Normal |
| Material to apply to a model | Prefix with “Mat\_” | Mat\_Checkpoint Red |
| Shader created user HLSL | Prefix with “Sh\_” | Sh\_Always on top |
| Shaders created with Shader Graph | Prefix with “Shg\_” | Shg\_Character Shader |
| Shader Sub Graph | Prefix with “Shg\_s\_” | Shg\_s\_Voronoi Noise |
| Prefab (This will make up most building blocks) | Prefix with “Pr\_” | Pr\_Bumper |
| A prefab only used as a VFX | Prefix with “Pr\_Effect\_” | Pr\_Effect\_General Explosion |
| A prefab related to a player character | Prefix with “PP\_” | PP\_Sonic |
| An enemy stored as a prefab | Prefix with “Pr\_E\_” | Pr\_E\_Motobug |
| A scriptable object, likely used for storing stats. | Prefix with “SO\_” | SO\_SonicStats |
| A prefab containing objects for UI | Prefix with “Pr\_UI\_” | Pr\_UI\_Boost |
| Audio Mixers | Prefix with “Mix\_” | Mix\_Music |
| Audio File not used for music | Prefix with “SE\_” | SE\_Bumper |
| Audio File used for music. | Prefix with “Mu\_” | Mu\_Stage Select |
| 3D object. FBX or OBJ | Prefix with “Ob\_” | Ob\_Monitor |
| Animation | Prefix with “An\_” | An\_Idle00 |
| Animation Controller | Prefix with “AC\_” | AC\_Sonic |
| Scene | Prefix with “Sc\_” | Sc\_Playground |

## Formatting

### Keywords -

For simplicity’s sake, frequently repeated conventions or terminology will be described here, rather than when used. Please note that there is a lot more to these styles than what is written here.

|  |  |  |
| --- | --- | --- |
| Keyword / Convention | Description | Example |
| Bracing | Enclosing sections of code between curly brackets {} (referred to as braces). Used in flow control and all of the “sections” in the previous table. |  |
| Indentation Style | How blocks of code are sorted in their “section”. E.G. the placement of the braces, where the lines start relevant to the header. |  |
| Allman style (indentation) | The block is indented once from the header, and braces are on their own lines. | **while** (x == y)  {  something();  } |
| K&R Style (Indentation) | The block is indented once, but the first brace is on the same line as the header. | **while** (x == y) {  something();  } |
| Pico Stye (Indentation) | The block is indented once, but the braces do not have their own lines. | **while** (x == y)  { something();  something\_else(); } |
| Inline formatting | AKA “single line” formatting. Where the code block is on the same line as the header. Braces are optional. | **If**(true) { something(); } |

### Indentation –

|  |  |  |
| --- | --- | --- |
| Section | Convention | Example |
| All | Unless otherwise stated, all blocks of code should be indented using Allman style. | **If**(true)  {  Something(); } |
| Methods | Use K&R Style.  This will make them easier to search for. | **If**(true) {  Something(); } |
| If statements that do nothing but “return;” if true. | Inline formatting  Do not remove braces. | **If**(true) { return; } |

### Spacing -

The following lists situations in which a space should or should not be used to separate commands and/or words. This does not mean when spaces are required to separate commands, but when having a space or not doesn’t matter to the compiler.

|  |  |  |
| --- | --- | --- |
| Section | Convention | Example |
| Setting a variable | Spaces around the = sign. | int i = 4; |
| Separating arguments | Spaces after comma | (int i, float f, string s) |
| Arguments when being set | Spaces at the start and end of brackets. | Method ( int i ) |
| Brackets in method header | Space before brackets | Void method() |
| Brackets when method is called | No space | Method() |
| Inside square brackets | No space | listOfNumbers[3] |
| Comparison operators | Spaces around the operator | f > i  f == i |
| Flow Control | No spaces at start and end of brackets. Space between command and brackets. | if (f > i)  while (i == f) |
| Class fields | Add indents between data type and variable name. | int i number; |

## Conventions

This section will detail certain rules followed for this project. This includes approaches to calculating certain features, frequently used methods, and more.

These are more specific to the context, and as such should not be applied to other code bases unless they have the same goal here.

### Calculating Velocity

Due to the flipper framework being all about the character's direction and speed, a major component is how that is calculated. How do environmental objects affect the character's speed? how does acceleration carry over into different states? How can speed limits be applied with so many elements increasing that speed? There are many considerations, and the answer arrived at is as follows.

Rather than using Unity’s character controller component, movement is directly set through the character's RigidBody component. While this is usually applied through the built-in AddForce methods, this takes control of the velocity directly, away from the user. They cannot adjust the direction for instance.

As such, the velocity is directly set in code, but if this was done across every script at every opportunity, it would make debugging difficult due to what caused the player velocity at the end of a frame being unclear, as it would depend on which was called last. Therefore,

Rule 1:  
 The RigidBody’s velocity should only be set at the end of the fixed update, by one line of code at the end of the PlayerPhysics script. All calculations should lead to this.

Because of this, velocity changes must be stored and only applied at the end of an Update. So, to handle this,

Rule 2:  
 When setting a new velocity, or adding another force to the current player’s velocity. Use the Add or Set Velocity methods all present together in the PlayerPhysics script.

Since every change to the velocity should go through these methods, debugging becomes a lot easier as using a Debug.Log provides notifications whenever velocity is changed, and a reference to what did it.

A problem with the original Bumper Engine, was that since everything came down to the same velocity calculations, environmental velocity would overlap constantly with controlled velocity. I.E. springs couldn’t launch players more than the max running speed due to clamping, dash rings would launch the player forwards and they would stay running at that same speed. So to counter this,

Rule 3:  
 The total velocity at the end is made up of core velocity (the velocity used in applying movement speeds, controlling animations and facing directions) and environmental velocity (the velocity added on top of the player, acquired from environmental objects.

The two types are seen frequently in the code base, where each frame, the player’s final velocity will be the product of both, but they are set by and will control different things. There are several advantages to this approach, the first being improved readability as the type of velocity shows the type of interactions. Another is how velocity can freely be applied to the player without messing up the character's animations (so they can move along a conveyor belt without their running speed increasing) or being limited by running clamps (springs can launch the player faster than running speed).

### Setting important values

Other key variables (such as player position), especially those that either affect many other elements when set (such as whether or not the player is grounded) or those that require additional checks when set (such as rolling due to changing the collider size), should also use methods for setting. This will make tracking calls to this easier, and decrease repetitions. So,

Rule 4:  
 Use designated “Set” methods when changing important variables, or those that directly have other effects when set.

### Action Rules

To decrease coupling, action or state scripts should not reference each other. Instead, any dependencies they have on other scripts, or changes they have depending on actions, should not check for each other, and instead use common public values stored in the Action Manager and Player Physics scripts. This includes Booleans for if air actions or certain controls are available, or the current state as an enum. Therefore,

Rule 5:  
 If an action behaves differently based on previous actions, do not reference those actions, instead, use a switch statement in the relevant action, that will use code based on the current player state variable in the action manager.

Rule 6:  
 Every main action script should have an accompanying constant in the player states enum, this should be used when calling ChangeAction.

Rule 7:  
 Each action when started or stopped, should set public variables in the main overhead scripts, rather than have those scripts check if this action is in use.

### Comments Rules

Comments are extremely important and should be used consistently. As such, there are rules for each category of comment.

Copyright Comments:  
 These include information about copyrights/licenses of source code files and are found at the beginning of the file. Due to this project being open source and from a variety of developers, do not implement these in framework scripts.

Header Comments:  
 These are included at the top of classes. Typically, these are used in team projects, as they provide additional information, such as the class author, revision number, or peer review status. These are currently not used in the project.

Member Comments:  
 These describe the functionality of and reasons for methods and fields/variables. Every method should have one of these in a line above the header to quickly give an overview of what it is for. Any important or unclear fields in the class should have comments that explain why they have been added, and what they track. If a serialized variable, use a ToolTip instead.

Inline Comments:  
 These are found in the body of the methods, and describe the implementation decisions of the relevant code. For simpler, single-line features, they can be on the same line, for groups of code, or flow control like if statements, they are one line above and describe the group of code below.

Section Comments:  
 These are used to separate sections of the code under headers. Currently, they are used to separate/organise different types of fields, the class structure (see class structure), and the different aspects actions should address when being started (such as visual effects, physics, controls, and action variables).

Code Comments:  
 These are lines of code temporarily “commented out” so their functionality is ignored but can easily be restored. These should only be used for debugging, and all instances have been removed from the project.

Copyright Comments:  
 Task comments. Developer notes containing todos, bug details, or other notes. These should be used in teams or during debugging, and there are a few in the current project to act as reminders or explanations when not directly linked to a specific line of code.

### Class Structure

Every refactored class is currently set to use regions and section comments to split the class into the following sections (if applicable).

* Properties (the global fields of the class)
  + Unity-specific properties (properties named using pascal case, see above).
  + Stats (serialized variables or ones set by the player stat objects).
  + Trackers (all other variables, that exist to keep track of data across playtime).
* Inherited Methods
  + MonoBehaviour (methods inherited from this, such as Start, Update, etc).
  + Interface (methods inherited from any attached interfaces.)
* Private Methods (methods only called by code in this class.)
* Public Methods (methods that can be called by other classes)
* Assigning (methods that set variables to external values such as stats in scriptable objects or shared assets. These should only be called once.)

The project files contain “templates”. Unused scripts that only exist to be copy pasted, so users can create new scripts within the structure.