Gamemakin UE4 Style Guide() {

A mostly reasonable approach to Unreal Engine 4

Heavily inspired by the Airbnb Javascript Style Guide.

Analytics lint partial support

Unreal Engine 4 Linter Plugin

An automated method of checking your project against this style guide is available for purchase at the Unreal Engine marketplace. This plugin's source code will eventually be free, but in order to use with UE4 without building the engine from source code, please use the marketplace version.

Linter and Style Guide Documentation

More technical documentation regarding Linter and the Style Guide can be found at our ReadTheDocs page.

Discuss This Style Guide

Gamemakin LLC has a public Discord channel at http://discord.gamemak.in with a #linter channel if you'd like to discuss all things style guide and Linter plugin.

Linking To This Document

Every section of this style guide is numbered for both easy reference and easy linking. You can link to any section directly by simply append a hash tag and the section number to the end of http://ue4.style For example, if you want to send someone to the first principle of this style guide you would append `#0.1`, resulting in http://ue4.style#0.1.

Forks And Translations

If you have made a notable fork or translation that is not suitable for a pull request into this repo, please submit a pull request to add the fork or translation here.

- Korean Translation by ymkim50
- Russian Translation by CosmoMyzrailGorynych
- Japanese Translation by akenatsu
- Chinese Translation by Beijing Skylens Tech.

Important Terminology

Levels/Maps

The word 'map' generally refers to what the average person calls a 'level' and may be used interchangeably. See this term's history here.

Cases

There are a few different ways you can name things. Here are some common casing types:

> PascalCase
>
> Capitalize every word and remove all spaces, e.g. `DesertEagle`, `StyleGuide`, `ASeriesOfWords`.
>
> camelCase
>
> The first letter is always lowercase but every following word starts with uppercase, e.g. `desertEagle`, `styleGuide`, `aSeriesOfWords`.
>
> Snake_case
>
> Words can arbitrarily start upper or lowercase but words are separated by an underscore, e.g. `desert_Eagle`, `Style_Guide`, `a_Series_of_Words`.

Variables / Properties

The words 'variable' and 'property' in most contexts are interchangable. If they are both used together in the same context however:

Property

Usually refers to a variable defined in a class. For example, if `BP_Barrel` had a variable `bExploded`, `bExploded` may be referred to as a property of `BP_Barrel`.

When in the context of a class, often used to imply accessing previously defined data.

Variable

Usually refers to a variable defined as a function argument or a local variable inside a function.

When in the context of a class, often used to convey discussion about its definition and what it will hold.

0. Principles

These principles have been adapted from idomatic.js style guide.

0.1 If your UE4 project already has a style guide, you should follow it.

If you are working on a project or with a team that has a pre-existing style guide, it should be respected. Any inconsistency between an existing style guide and this guide should defer to the existing.

Style guides should be living documents however and you should propose style guide changes to an existing style guide as well as this guide if you feel the change benefits all usages.

> "Arguments over style are pointless. There should be a style guide, and you should follow it."
>
> *Rebecca Murphey*

0.2 All structure, assets, and code in any Unreal Engine 4 project should look like a single person created it, no matter how many people contributed.

Moving from one project to another should not cause a re-learning of style and structure. Conforming to a style guide removes unneeded guesswork and ambiguities.

It also allows for more productive creation and maintenance as one does not need to think about style, simply follow instructions. This style guide is written with best practices in mind, meaning that by following this style guide you will also minimize hard to track issues.

0.3 Friends do not let friends have bad style.

If you see someone working either against a style guide or no style guide, try to correct them.

When working within a team or discussing within a community such as Unreal Slackers, it is far easier to help and to ask for help when people are consistent. Nobody likes to help untangle someone's Blueprint spaghetti or deal with assets with names they can't understand.

If you are helping someone who's work conforms to a different but consistent and sane style guide, you should be able to adapt to it. If they do not conform to any style guide, please direct them here.

0.4 A team without a style guide is no team of mine.

When joining an Unreal Engine 4 team one of your first questions should be "Do you have a style guide?". If the answer is no, you should be skeptical about their ability to work as a team.

0.5 Don't Break The Law

Gamemakin LLC is not a lawyer, but please don't introduce illegal actions and behavior to a project, including but not limited to:

- Don't distribute content you don't have the rights to distribute
- Don't infringe on someone else's copyrighted or trademark material
- Don't steal content
- Follow licensing restrictions on content, e.g. attribute when attributions are needed

Table of Contents

1. Asset Naming Conventions
2. Directory Structure
3. Blueprints
4. Static Meshes
5. Particle Systems
6. Levels / Maps
7. Textures

1. Asset Naming Conventions lint partial support

Naming conventions should be treated as law. A project that conforms to a naming convention is able to have its assets managed, searched, parsed, and maintained with incredible ease.

Most things are prefixed with prefixes being generally an acronym of the asset type followed by an underscore.

1.1 Base Asset Name - `Prefix_BaseAssetName_Variant_Suffix` lint partial support

All assets should have a *Base Asset Name*. A Base Asset Name represents a logical grouping of related assets. Any asset that is part of this logical group should follow the standard of `Prefix_BaseAssetName_Variant_Suffix`.

Keeping the pattern `Prefix_BaseAssetName_Variant_Suffix` and in mind and using common sense is generally enough to warrant good asset names. Here are some detailed rules regarding each element.

`Prefix` and `Suffix` are to be determined by the asset type through the following Asset Name Modifier tables.

`BaseAssetName` should be determined by a short and easily recognizable name related to the context of this group of assets. For example, if you had a character named Bob, all of Bob's assets would have the `BaseAssetName` of `Bob`.

For unique and specific variations of assets, `Variant` is either a short and easily recognizable name that represents logical grouping of assets that are a subset of an asset's base name. For example, if Bob had multiple skins these skins should still use `Bob` as the `BaseAssetName` but include a recognizable `Variant`. An 'Evil' skin would be referred to as `Bob_Evil` and a 'Retro' skin would be referred to as `Bob_Retro`.

For unique but generic variations of assets, `Variant` is a two digit number starting at `01`. For example, if you have an environment artist generating nondescript rocks, they would be named `Rock_01`, `Rock_02`, `Rock_03`, etc. Except for rare exceptions, you should never require a three digit variant number. If you have more than 100 assets, you should consider organizing them with different base names or using multiple variant names.

Depending on how your asset variants are made, you can chain together variant names. For example, if you are creating flooring assets for an Arch Viz project you should use the base name `Flooring` with chained variants such as `Flooring_Marble_01`, `Flooring_Maple_01`, `Flooring_Tile_Squares_01`.

1.1 Examples

1.1e1 Bob

Asset Type	Asset Name
Skeletal Mesh	SK_Bob
Material	M_Bob
Texture (Diffuse/Albedo)	T_Bob_D
Texture (Normal)	T_Bob_N
Texture (Evil Diffuse)	T_Bob_Evil_D

1.1e2 Rocks

Asset Type	Asset Name
Static Mesh (01)	S_Rock_01
Static Mesh (02)	S_Rock_02
Static Mesh (03)	S_Rock_03
Material	M_Rock
Material Instance (Snow)	MI_Rock_Snow

1.2 Asset Name Modifiers lint supported

When naming an asset use these tables to determine the prefix and suffix to use with an asset's Base Asset Name.

Sections

- 1.2.1 Most Common
- 1.2.2 Animations
- 1.2.3 Artificial Intelligence
- 1.2.4 Blueprints
- 1.2.5 Materials
- 1.2.6 Textures
- 1.2.7 Miscellaneous
- 1.2.8 Paper 2D
- 1.2.9 Physics
- 1.2.10 Sound
- 1.2.11 User Interface
- 1.2.12 Effects

1.2.1 Most Common lint supported

Asset Type	Prefix	Suffix	Notes
Level / Map			Should be in a folder called Maps.
Level (Persistent)		_P	
Level (Audio)		_Audio	
Level (Lighting)		_Lighting	
Level (Geometry)		_Geo	
Level (Gameplay)		_Gameplay	
Blueprint	BP_		
Material	M_		
Static Mesh	S_		Many use SM_. We use S_.
Skeletal Mesh	SK_		
Texture	T_	_?	See Textures
Particle System	PS_		
Widget Blueprint	WBP_		

1.2.2 Animations lint supported

Asset Type	Prefix	Suffix	Notes
Aim Offset	AO_		
Aim Offset 1D	AO_		
Animation Blueprint	ABP_		
Animation Composite	AC_		
Animation Montage	AM_		
Animation Sequence	A_		
Blend Space	BS_		
Blend Space 1D	BS_		
Level Sequence	LS_		
Morph Target	MT_		
Paper Flipbook	PFB_		
Rig	Rig_		
Skeletal Mesh	SK_		
Skeleton	SKEL_		

1.2.3 Artificial Intelligence lint supported

Asset Type	Prefix	Suffix	Notes
AI Controller	AIC_		
Behavior Tree	BT_		
Blackboard	BB_		
Decorator	BTDecorator_		
Service	BTService_		
Task	BTTask_		
Environment Query	EQS_		
EnvQueryContext	EQS_	Context	

1.2.4 Blueprints lint supported

Asset Type	Prefix	Suffix	Notes
Blueprint	BP_		
Blueprint Component	BP_	Component	I.e. BP_InventoryComponent
Blueprint Function Library	BPFL_		
Blueprint Interface	BPI_		
Blueprint Macro Library	BPML_		Do not use macro libraries if possible.
Enumeration	E		No underscore.
Structure	F or S		No underscore.
Tutorial Blueprint	TBP_		
Widget Blueprint	WBP_		

1.2.5 Materials lint supported

Asset Type	Prefix	Suffix	Notes
Material	M_		
Material (Post Process)	PP_		
Material Function	MF_		
Material Instance	MI_		
Material Parameter Collection	MPC_		
Subsurface Profile	SP_		
Physical Materials	PM_		
Decal	M_, MI_	_Decal	

1.2.6 Textures lint supported

Asset Type	Prefix	Suffix	Notes
Texture	T_		
Texture (Diffuse/Albedo/Base Color)	T_	_D	
Texture (Normal)	T_	_N	
Texture (Roughness)	T_	_R	
Texture (Alpha/Opacity)	T_	_A	
Texture (Ambient Occlusion)	T_	_O	
Texture (Bump)	T_	_B	
Texture (Emissive)	T_	_E	
Texture (Mask)	T_	_M	
Texture (Specular)	T_	_S	
Texture (Metallic)	T_	_M	
Texture (Packed)	T_	_*	See notes below about packing.
Texture Cube	TC_		
Media Texture	MT_		
Render Target	RT_		
Cube Render Target	RTC_		
Texture Light Profile	TLP		

1.2.6.1 Texture Packing lint unsupported

It is common practice to pack multiple layers of texture data into one texture. An example of this is packing Emissive, Roughness, Ambient Occlusion together as the Red, Green, and Blue channels of a texture respectively. To determine the suffix, simply stack the given suffix letters from above together, e.g. `_ERO`.

> It is generally acceptable to include an Alpha/Opacity layer in your Diffuse/Albedo's alpha channel and as this is common practice, adding `A` to the `_D` suffix is optional.

Packing 4 channels of data into a texture (RGBA) is not recommended except for an Alpha/Opacity mask in the Diffuse/Albedo's alpha channel as a texture with an alpha channel incurs more overhead than one without.

1.2.7 Miscellaneous lint supported

Asset Type	Prefix	Suffix	Notes
Animated Vector Field	VFA_		
Camera Anim	CA_		
Color Curve	Curve_	_Color	
Curve Table	Curve_	_Table	
Data Asset	*_		Prefix should be based on class.
Data Table	DT_		
Float Curve	Curve_	_Float	
Foliage Type	FT_		
Force Feedback Effect	FFE_		
Landscape Grass Type	LG_		
Landscape Layer	LL_		
Matinee Data	Matinee_		
Media Player	MP_		
Object Library	OL_		
Redirector			These should be fixed up ASAP.
Sprite Sheet	SS_		
Static Vector Field	VF_		
Substance Graph Instance	SGI_		
Substance Instance Factory	SIF_		
Touch Interface Setup	TI_		
Vector Curve	Curve_	_Vector	

1.2.8 Paper 2D lint supported

Asset Type	Prefix	Suffix	Notes
Paper Flipbook	PFB_		
Sprite	SPR_		
Sprite Atlas Group	SPRG_		
Tile Map	TM_		
Tile Set	TS_		

1.2.9 Physics lint supported

Asset Type	Prefix	Suffix	Notes
Physical Material	PM_		
Physical Asset	PHYS_		
Destructible Mesh	DM_		

1.2.10 Sounds lint supported

Asset Type	Prefix	Suffix	Notes
Dialogue Voice	DV_		
Dialogue Wave	DW_		
Media Sound Wave	MSW_		
Reverb Effect	Reverb_		
Sound Attenuation	ATT_		
Sound Class			No prefix/suffix. Should be put in a folder called SoundClasses
Sound Concurrency		_SC	Should be named after a SoundClass
Sound Cue	A_	_Cue	
Sound Mix	Mix_		
Sound Wave	A_		

1.2.11 User Interface lint supported

Asset Type	Prefix	Suffix	Notes
Font	Font_		
Slate Brush	Brush_		
Slate Widget Style	Style_		
Widget Blueprint	WBP_		

1.2.12 Effects lint supported

Asset Type	Prefix	Suffix	Notes
Particle System	PS_		
Material (Post Process)	PP_		

Back to Top

2. Content Directory Structure lint partial support

Equally important as asset names, the directory structure style of a project should be considered law. Asset naming conventions and content directory structure go hand in hand, and a violation of either causes unneeded chaos.

There are multiple ways to lay out the content of a UE4 project. In this style, we will be using a structure that relies more on filtering and search abilities of the Content Browser for those working with assets to find assets of a specific type instead of another common structure that groups asset types with folders.

> If you are using the prefix naming convention above, using folders to contain assets of similar types such as `Meshes`, `Textures`, and `Materials` is a redundant practice as asset types are already both sorted by prefix as well as able to be filtered in the content browser.

2e1 Example Project Content Structure

```
|-- Content
    |-- GenericShooter
        |-- Art
        |   |-- Industrial
        |   |   |-- Ambient
        |   |   |-- Machinery
        |   |   |-- Pipes
        |   |-- Nature
        |   |   |-- Ambient
        |   |   |-- Foliage
        |   |   |-- Rocks
        |   |   |-- Trees
        |   |-- Office
        |-- Characters
        |   |-- Bob
        |   |-- Common
        |   |   |-- Animations
        |   |   |-- Audio
        |   |-- Jack
        |   |-- Steve
        |   |-- Zoe
        |-- Core
        |   |-- Characters
        |   |-- Engine
        |   |-- GameModes
        |   |-- Interactables
        |   |-- Pickups
        |   |-- Weapons
        |-- Effects
        |   |-- Electrical
        |   |-- Fire
        |   |-- Weather
        |-- Maps
        |   |-- Campaign1
        |   |-- Campaign2
        |-- MaterialLibrary
        |   |-- Debug
        |   |-- Metal
        |   |-- Paint
        |   |-- Utility
        |   |-- Weathering
        |-- Placeables
        |   |-- Pickups
        |-- Weapons
            |-- Common
            |-- Pistols
            |   |-- DesertEagle
            |   |-- RocketPistol
            |-- Rifles
```

The reasons for this structure are listed in the following sub-sections.

Sections

- 2.1 Folder Names
- 2.2 Top-Level Folders
- 2.3 Developer Folders
- 2.4 Maps
- 2.5 Core
- 2.6 `Assets` and `AssetTypes`
- 2.7 Large Sets
- 2.8 Material Library

2.1 Folder Names lint partial support

These are common rules for naming any folder in the content structure.

2.1.1 Always Use PascalCase* lint supported

PascalCase refers to starting a name with a capital letter and then instead of using spaces, every following word also starts with a capital letter. For example, `DesertEagle`, `RocketPistol`, and `ASeriesOfWords`.

See Cases.

2.1.2 Never Use Spaces lint supported

Re-enforcing 2.1.1, never use spaces. Spaces can cause various engineering tools and batch processes to fail. Ideally, your project's root also contains no spaces and is located somewhere such as `D:\Project` instead of `C:\Users\My Name\My Documents\Unreal Projects`.

2.1.3 Never Use Unicode Characters And Other Symbols lint supported

If one of your game characters is named 'Zoë', its folder name should be `Zoe`. Unicode characters can be worse than Spaces for engineering tool and some parts of UE4 don't support Unicode characters in paths either.

Related to this, if your project has unexplained issues and your computer's user name has a Unicode character (i.e. your name is `Zoë`), any project located in your `My Documents` folder will suffer from this issue. Often simply moving your project to something like `D:\Project` will fix these mysterious issues.

Using other characters outside `a-z`, `A-Z`, and `0-9` such as `@`, `-`, `_`, `,`, `*`, and `#` can also lead to unexpected and hard to track issues on other platforms, source control, and weaker engineering tools.

2.2 Use A Top Level Folder For Project Specific Assets lint supported

All of a project's assets should exist in a folder named after the project. For example, if your project is named 'Generic Shooter', all of it's content should exist in `Content/GenericShooter`.

> The `Developers` folder is not for assets that your project relies on and therefore is not project specific. See Developer Folders for details about this.

There are multiple reasons for this approach.

2.2.1 No Global Assets

Often in code style guides it is written that you should not pollute the global namespace and this follows the same principle. When assets are allowed to exist outside of a project folder it often becomes much harder to enforce a strict structure layout as assets not in a folder encourages the bad behavior of not having to organize assets.

Every asset should have a purpose, otherwise it does not belong in a project. If an asset is an experimental test and shouldn't be used by the project it should be put in a `Developer` folder.

2.2.2 Reduce Migration Conflicts

When working on multiple projects it is common for a team to copy assets from one project to another if they have made something useful for both. When this occurs, the easiest way to perform the copy is to use the Content Browser's Migrate functionality as it will copy over not just the selected asset but all of its dependencies.

These dependencies are what can easily get you into trouble. If two project's assets do not have a top level folder and they happen to have similarly named or already previously migrated assets, a new migration can accidentally wipe any changes to the existing assets.

This is also the primary reason why Epic's Marketplace staff enforces the same policy for submitted assets.

After a migration, safe merging of assets can be done using the 'Replace References' tool in the content browser with the added clarity of assets not belonging to a project's top level folder are clearly pending a merge. Once assets are merged and fully migrated, there shouldn't be another top level folder in your Content tree. This method is *100%* guaranteed to make any migrations that occur completely safe.

2.2.2e1 Master Material Example

For example, say you created a master material in one project that you would like to use in another project so you migrated that asset over. If this asset is not in a top level folder, it may have a name like `Content/MaterialLibrary/M_Master`. If the target project doesn't have a master material already, this should work without issue.

As work on one or both projects progress their respective master materials may change to be tailored for their specific projects due to the course of normal development.

The issue comes when, for example, an artist for one project created a nice generic modular set of static meshes and someone wants to include that set of static meshes in the second project. If the artist who created the assets used material instances based on `Content/MaterialLibrary/M_Master` as they're instructed to, when a migration is performed there is a great chance of conflict for the previously migrated `Content/MaterialLibrary/M_Master` asset.

This issue can be hard to predict and hard to account for. The person migrating the static meshes may not be the same person who is familiar with the development of both project's master material, and they may not be even aware that the static meshes in question rely on material instances which then rely on the master material. The Migrate tool requires the entire chain of dependencies to work however, and so it will be forced to grab `Content/MaterialLibrary/M_Master` when it copies these assets to the other project and it will overwrite the existing asset.

It is at this point where if the master materials for both projects are incompatible in *any way*, you risk breaking possibly the entire material library for a project as well as any other dependencies that may have already been migrated, simply because assets were not stored in a top level folder. The simple migration of static meshes now becomes a very ugly task.

2.2.3 Samples, Templates, and Marketplace Content Are Risk-Free

An extension to 2.2.2, if a team member decides to add sample content, template files, or assets they bought from the marketplace, it is guaranteed, as long your project's top-level folder is uniquely named, these new assets will not interfere with your project.

You can not trust marketplace content to fully conform to the top level folder rule. There exist many assets that have the majority of their content in a top level folder but also have possibly modified Epic sample content as well as level files polluting the global `Content` folder.

When adhering to 2.2, the worst marketplace conflict you can have is if two marketplace assets both have the same Epic sample content. If all your assets are in a project specific folder, including sample content you may have moved into your folder, your project will never break.

2.2.4 DLC, Sub-Projects, and Patches Are Easily Maintained

If your project plans to release DLC or has multiple sub-projects associated with it that may either be migrated out or simply not cooked in a build, assets relating to these projects should have their own separate top level content folder. This make cooking DLC separate from main project content far easier. Sub-projects can also be migrated in and out with minimal effort. If you need to change a material of an asset or add some very specific asset override behavior in a patch, you can easily put these changes in a patch folder and work safely without the chance of breaking the core project.

2.3 Use Developers Folder For Local Testing lint unsupported

During a project's development, it is very common for team members to have a sort of 'sandbox' where they can experiment freely without risking the core project. Because this work may be ongoing, these team members may wish to put their assets on a project's source control server. Not all teams require use of Developer folders, but ones that do use them often run into a common problem with assets submitted to source control.

It is very easy for a team member to accidentally use assets that are not ready for use which will cause issues once those assets are removed. For example, an artist may be iterating on a modular set of static meshes and still working on getting their sizing and grid snapping correct. If a world builder sees these assets in the main project folder, they might use them all over a level not knowing they could be subject to incredible change and/or removal. This causes massive amounts of re-working by everyone on the team to resolve.

If these modular assets were placed in a Developer folder, the world builder should never of had a reason to use them and the whole issue would never happen. The Content Browser has specific View Options that will hide Developer folders (they are hidden by default) making it impossible to accidentally use Developer assets under normal use.

Once the assets are ready for use, an artist simply has to move the assets into the project specific folder and fix up redirectors. This is essentially 'promoting' the assets from experimental to production.

2.4 All Map* Files Belong In A Folder Called Maps lint supported

Map files are incredibly special and it is common for every project to have its own map naming system, especially if they work with sub-levels or streaming levels. No matter what system of map organization is in place for the specific project, all levels should belong in `/Content/Project/Maps`.

Being able to tell someone to open a specific map without having to explain where it is is a great time saver and general 'quality of life' improvement. It is common for levels to be within sub-folders of `Maps`, such as `Maps/Campaign1/` or `Maps/Arenas`, but the most important thing here is that they all exist within `/Content/Project/Maps`.

This also simplifies the job of cooking for engineers. Wrangling levels for a build process can be extremely frustrating if they have to dig through arbitrary folders for them. If a team's maps are all in one place, it is much harder to accidentally not cook a map in a build. It also simplifies lighting build scripts as well as QA processes.

2.5 Use A `Core` Folder For Critical Blueprints And Other Assets lint unsupported

Use `/Content/Project/Core` folder for assets that are absolutely fundamental to a project's workings. For example, base `GameMode`, `Character`, `PlayerController`, `GameState`, `PlayerState`, and related Blueprints should live here.

This creates a very clear "don't touch these" message for other team members. Non-engineers should have very little reason to enter the `Core` folder. Following good code structure style, designers should be making their gameplay tweaks in child classes that expose functionality. World builders should be using prefab Blueprints in designated folders instead of potentially abusing base classes.

For example, if your project requires pickups that can be placed in a level, there should exist a base Pickup class in `Core/Pickups` that defines base behavior for a pickup. Specific pickups such as a Health or Ammo should exist in a folder such as `/Content/Project/Placeables/Pickups/`. Game designers can define and tweak pickups in this folder however they please, but they should not touch `Core/Pickups` as they may unintentionally break pickups project-wide.

2.6 Do Not Create Folders Called `Assets` or `AssetTypes` lint supported

2.6.1 Creating a folder named `Assets` is redundant. lint supported

All assets are assets.

2.6.2 Creating a folder named `Meshes`, `Textures`, or `Materials` is redundant. lint supported

All asset names are named with their asset type in mind. These folders offer only redundant information and the use of these folders can easily be replaced with the robust and easy to use filtering system the Content Browser provides.

Want to view only static mesh in `Environment/Rocks/`? Simply turn on the Static Mesh filter. If all assets are named correctly, they will also be sorted in alphabetical order regardless of prefixes. Want to view both static meshes and skeletal meshes? Simply turn on both filters. This eliminates the need to potentially have to `Control-Click` select two folders in the Content Browser's tree view.

> This also extends the full path name of an asset for very little benefit. The `S_` prefix for a static mesh is only two characters, whereas `Meshes/` is seven characters.

Not doing this also prevents the inevitability of someone putting a static mesh or a texture in a `Materials` folder.

2.7 Very Large Asset Sets Get Their Own Folder Layout lint unsupported

This can be seen as a pseudo-exception to 2.6.

There are certain asset types that have a huge volume of related files where each asset has a unique purpose. The two most common are Animation and Audio assets. If you find yourself having 15+ of these assets that belong together, they should be together.

For example, animations that are shared across multiple characters should lay in `Characters/Common/Animations` and may have sub-folders such as `Locomotion` or `Cinematic`.

> This does not apply to assets like textures and materials. It is common for a `Rocks` folder to have a large amount of textures if there are a large amount of rocks, however these textures are generally only related to a few specific rocks and should be named appropriately. Even if these textures are part of a Material Library.

2.8 `MaterialLibrary` lint unsupported

If your project makes use of master materials, layered materials, or any form of reusable materials or textures that do not belong to any subset of assets, these assets should be located in `Content/Project/MaterialLibrary`.

This way all 'global' materials have a place to live and are easily located.

> This also makes it incredibly easy to enforce a 'use material instances only' policy within a project. If all artists and assets should be using material instances, then the only regular material assets that should exist are within this folder. You can easily verify this by searching for base materials in any folder that isn't the `MaterialLibrary`.

The `MaterialLibrary` doesn't have to consist of purely materials. Shared utility textures, material functions, and other things of this nature should be stored here as well within folders that designate their intended purpose. For example, generic noise textures should be located in `MaterialLibrary/Utility`.

Any testing or debug materials should be within `MaterialLibrary/Debug`. This allows debug materials to be easily stripped from a project before shipping and makes it incredibly apparent if production assets are using them if reference errors are shown.

2.9 No Empty Folders lint unsupported

There simply shouldn't be any empty folders. They clutter the content browser.

If you find that the content browser has an empty folder you can't delete, you should perform the following:

1. Be sure you're using source control.
2. Immediately run Fix Up Redirectors on your project.
3. Navigate to the folder on-disk and delete the assets inside.
4. Close the editor.
5. Make sure your source control state is in sync (i.e. if using Perforce, run a Reconcile Offline Work on your content directory)
6. Open the editor. Confirm everything still works as expected. If it doesn't, revert, figure out what went wrong, and try again.
7. Ensure the folder is now gone.
8. Submit changes to source control.

Back to Top

3. Blueprints lint partial support

This section will focus on Blueprint classes and their internals. When possible, style rules conform to Epic's Coding Standard.

Remember: Blueprinting badly bears blunders, beware! (Phrase by KorkuVeren)

Sections

- 3.1 Compiling
- 3.2 Variables
- 3.3 Functions
- 3.4 Graphs

3.1 Compiling lint supported

All blueprints should compile with zero warnings and zero errors. You should fix blueprint warnings and errors immediately as they can quickly cascade into very scary unexpected behavior.

Do *not* submit broken blueprints to source control. If you must store them on source control, shelve them instead.

Broken blueprints can cause problems that manifest in other ways, such as broken references, unexpected behavior, cooking failures, and frequent unneeded recompilation. A broken blueprint has the power to break your entire game.

3.2 Variables lint partial support

The words `variable` and `property` may be used interchangeably.

Sections

- 3.2.1 Naming
- 3.2.2 Editable
- 3.2.3 Categories
- 3.2.4 Access
- 3.2.5 Advanced
- 3.2.6 Transient
- 3.2.7 Config

3.2.1 Naming lint partial support

3.2.1.1 Nouns lint unsupported

All non-boolean variable names must be clear, unambiguous, and descriptive nouns.

3.2.1.2 PascalCase lint supported

All non-boolean variables should be in the form of PascalCase.

3.2.1.2e Examples:

- `Score`
- `Kills`
- `TargetPlayer`
- `Range`
- `CrosshairColor`
- `AbilityID`

3.2.1.3 Boolean `b` Prefix lint supported

All booleans should be named in PascalCase but prefixed with a lowercase `b`.

Example: Use `bDead` and `bEvil`, **not** `Dead` and `Evil`.

UE4 Blueprint editors know not to include the `b` in user-friendly displays of the variable.

3.2.1.4 Boolean Names lint partial support

3.2.1.4.1 General And Independent State Information lint supported

All booleans should be named as descriptive adjectives when possible if representing general information. Do not include words that phrase the variable as a question, such as `Is`. This is reserved for functions.

Example: Use `bDead` and `bHostile` **not** `bIsDead` and `bIsHostile`.

Try to not use verbs such as `bRunning`. Verbs tend to lead to complex states.

3.2.1.4.2 Complex States lint unsupported

Do not to use booleans to represent complex and/or dependent states. This makes state adding and removing complex and no longer easily readable. Use an enumeration instead.

Example: When defining a weapon, do **not** use `bReloading` and `bEquipping` if a weapon can't be both reloading and equipping. Define an enumeration named `EWeaponState` and use a variable with this type named `WeaponState` instead. This makes it far easier to add new states to weapons.

Example: Do **not** use `bRunning` if you also need `bWalking` or `bSprinting`. This should be defined as an enumeration with clearly defined state names.

3.2.1.5 Considered Context lint unsupported

All variable names must not be redundant with their context as all variable references in Blueprint will always have context.

3.2.1.5e Examples:

Consider a Blueprint called `BP_PlayerCharacter`.

Bad

- `PlayerScore`
- `PlayerKills`
- `MyTargetPlayer`
- `MyCharacterName`
- `CharacterSkills`
- `ChosenCharacterSkin`

All of these variables are named redundantly. It is implied that the variable is representative of the `BP_PlayerCharacter` it belongs to because it is `BP_PlayerCharacter` that is defining these variables.

Good

- `Score`
- `Kills`
- `TargetPlayer`
- `Name`
- `Skills`
- `Skin`

3.2.1.6 Do *Not* Include Atomic Type Names lint supported

Atomic or primitive variables are variables that represent data in their simplest form, such as booleans, integers, floats, and enumerations.

Strings and vectors are considered atomic in terms of style when working with Blueprints, however they are technically not atomic.

> While vectors consist of three floats, vectors are often able to be manipulated as a whole, same with rotators.

> Do *not* consider Text variables as atomic, they are secretly hiding localization functionality. The atomic type of a string of characters is `String`, not `Text`.

Atomic variables should not have their type name in their name.

Example: Use `Score`, `Kills`, and `Description` **not** `ScoreFloat`, `FloatKills`, `DescriptionString`.

The only exception to this rule is when a variable represents 'a number of' something to be counted *and* when using a name without a variable type is not easy to read.

Example: A fence generator needs to generate X number of posts. Store X in `NumPosts` or `PostsCount` instead of `Posts` as `Posts` may potentially read as an Array of a variable type named `Post`.

3.2.1.7 Do Include Non-Atomic Type Names lint unsupported

Non-atomic or complex variables are variables that represent data as a collection of atomic variables. Structs, Classes, Interfaces, and primitives with hidden behavior such as `Text` and `Name` all qualify under this rule.

> While an Array of an atomic variable type is a list of variables, Arrays do not change the 'atomicness' of a variable type.

These variables should include their type name while still considering their context.

If a class owns an instance of a complex variable, i.e. if a `BP_PlayerCharacter` owns a `BP_Hat`, it should be stored as the variable type as without any name modifications.

Example: Use `Hat`, `Flag`, and `Ability` **not** `MyHat`, `MyFlag`, and `PlayerAbility`.

If a class does not own the value a complex variable represents, you should use a noun along with the variable type.

Example: If a `BP_Turret` has the ability to target a `BP_PlayerCharacter`, it should store its target as `TargetPlayer` as when in the context of `BP_Turret` it should be clear that it is a reference to another complex variable type that it does not own.

3.2.1.8 Arrays lint partial support

Arrays follow the same naming rules as above, but should be named as a plural noun.

Example: Use `Targets`, `Hats`, and `EnemyPlayers`, **not** `TargetList`, `HatArray`, `EnemyPlayerArray`.

3.2.2 Editable Variables lint partial support

All variables that are safe to change the value of in order to configure behavior of a blueprint should be marked as `Editable`.

Conversely, all variables that are not safe to change or should not be exposed to designers should *not* be marked as editable, unless for engineering reasons the variable must be marked as `Expose On Spawn`.

Do not arbitrarily mark variables as `Editable`.

3.2.2.1 Tooltips lint supported

All `Editable` variables, including those marked editable just so they can be marked as `Expose On Spawn`, should have a description in their `Tooltip` fields that explains how changing this value affects the behavior of the blueprint.

3.2.2.2 Slider And Value Ranges lint unsupported

All `Editable` variables should make use of slider and value ranges if there is ever a value that a variable should *not* be set to.

Example: A blueprint that generates fence posts might have an editable variable named `PostsCount` and a value of -1 would not make any sense. Use the range fields to mark 0 as a minimum.

If an editable variable is used in a Construction Script, it should have a reasonable Slider Range defined so that someone can not accidentally assign it a large value that could crash the editor.

A Value Range only needs to be defined if the bounds of a value are known. While a Slider Range prevents accidental large number inputs, an undefined Value Range allows a user to specify a value outside the Slider Range that may be considered 'dangerous' but still valid.

3.2.3 Categories lint supported

If a class has only a small number of variables, categories are not required.

If a class has a moderate amount of variables (5-10), all `Editable` variables should have a non-default category assigned. A common category is `Config`.

If a class has a large amount of variables, all `Editable` variables should be categorized into sub-categories using the category `Config` as the base category. Non-editable variables should be categorized into descriptive categories describing their usage.

> You can define sub-categories by using the pipe character `|`, i.e. `Config | Animations`.

Example: A weapon class set of variables might be organized as:

```
|-- Config
|     |-- Animations
|     |-- Effects
|     |-- Audio
|     |-- Recoil
|     |-- Timings
|-- Animations
|-- State
|-- Visuals
```

3.2.4 Variable Access Level lint unsupported

In C++, variables have a concept of access level. Public means any code outside the class can access the variable. Protected means only the class and any child classes can access this variable internally. Private means only this class and no child classes can access this variable.

Blueprints do not have a defined concept of protected access currently.

Treat `Editable` variables as public variables. Treat non-editable variables as protected variables.

3.2.4.1 Private Variables lint unsupported

Unless it is known that a variable should only be accessed within the class it is defined and never a child class, do not mark variables as private. Until variables are able to be marked `protected`, reserve private for when you absolutely know you want to restrict child class usage.

3.2.5 Advanced Display lint unsupported

If a variable should be editable but often untouched, mark it as `Advanced Display`. This makes the variable hidden unless the advanced display arrow is clicked.

To find the `Advanced Display` option, it is listed as an advanced displayed variable in the variable details list.

3.2.6 Transient Variables lint unsupported

Transient variables are variables that do not need to have their value saved and loaded and have an initial value of zero or null. This is useful for references to other objects and actors who's value isn't known until run-time. This prevents the editor from ever saving a reference to it, and speeds up saving and loading of the blueprint class.

Because of this, all transient variables should always be initialized as zero or null. To do otherwise would result in hard to debug errors.

3.2.8 Config Variables lint supported

Do not use the `Config Variable` flag. This makes it harder for designers to control blueprint behavior. Config variables should only be used in C++ for rarely changed variables. Think of them as `Advanced Advanced Display` variables.

3.3 Functions, Events, and Event Dispatchers lint unsupported

This section describes how you should author functions, events, and event dispatchers. Everything that applies to functions also applies to events, unless otherwise noted.

3.3.1 Function Naming lint unsupported

The naming of functions, events, and event dispatchers is critically important. Based on the name alone, certain assumptions can be made about functions. For example:

- Is it a pure function?
- Is it fetching state information?
- Is it a handler?
- Is it an RPC?
- What is its purpose?

These questions and more can all be answered when functions are named appropriately.

3.3.1.1 All Functions Should Be Verbs lint unsupported

All functions and events perform some form of action, whether its getting info, calculating data, or causing something to explode. Therefore, all functions should all start with verbs. They should be worded in the present tense whenever possible. They should also have some context as to what they are doing.

`OnRep` functions, event handlers, and event dispatchers are an exception to this rule.

Good examples:

- `Fire` - Good example if in a Character / Weapon class, as it has context. Bad if in a Barrel / Grass / any ambiguous class.
- `Jump` - Good example if in a Character class, otherwise, needs context.
- `Explode`
- `ReceiveMessage`
- `SortPlayerArray`
- `GetArmOffset`
- `GetCoordinates`
- `UpdateTransforms`
- `EnableBigHeadMode`
- `IsEnemy` - "Is" is a verb.

Bad examples:

- `Dead` - Is Dead? Will deaden?
- `Rock`
- `ProcessData` - Ambiguous, these words mean nothing.
- `PlayerState` - Nouns are ambiguous.
- `Color` - Verb with no context, or ambiguous noun.

3.3.1.2 Property RepNotify Functions Always `OnRep_Variable` lint unsupported

All functions for replicated with notification variables should have the form `OnRep_Variable`. This is forced by the Blueprint editor. If you are writing a C++ `OnRep` function however, it should also follow this convention when exposing it to Blueprints.

3.3.1.3 Info Functions Returning Bool Should Ask Questions lint unsupported

When writing a function that does not change the state of or modify any object and is purely for getting information, state, or computing a yes/no value, it should ask a question. This should also follow the verb rule.

This is extremely important as if a question is not asked, it may be assumed that the function performs an action and is returning whether that action succeeded.

Good examples:

- `IsDead`
- `IsOnFire`
- `IsAlive`
- `IsSpeaking`
- `IsHavingAnExistentialCrisis`
- `IsVisible`
- `HasWeapon` - "Has" is a verb.
- `WasCharging` - "Was" is past-tense of "be". Use "was" when referring to 'previous frame' or 'previous state'.
- `CanReload` - "Can" is a verb.

Bad examples:

- `Fire` - Is on fire? Will fire? Do fire?
- `OnFire` - Can be confused with event dispatcher for firing.
- `Dead` - Is dead? Will deaden?
- `Visibility` - Is visible? Set visibility? A description of flying conditions?

3.3.1.4 Event Handlers and Dispatchers Should Start With `On` lint unsupported

Any function that handles an event or dispatches an event should start with `On` and continue to follow the verb rule. The verb may move to the end however if past-tense reads better.

Collocations of the word `On` are exempt from following the verb rule.

`Handle` is not allowed. It is 'Unreal' to use `On` instead of `Handle`, while other frameworks may prefer to use `Handle` instead of `On`.

Good examples:

- `OnDeath` - Common collocation in games
- `OnPickup`
- `OnReceiveMessage`
- `OnMessageRecieved`
- `OnTargetChanged`
- `OnClick`
- `OnLeave`

Bad examples:

- `OnData`
- `OnTarget`
- `HandleMessage`
- `HandleDeath`

3.3.1.5 Remote Procedure Calls Should Be Prefixed With Target lint unsupported

Any time an RPC is created, it should be prefixed with either `Server`, `Client`, or `Multicast`. No exceptions.

After the prefix, follow all other rules regarding function naming.

Good examples:

- `ServerFireWeapon`
- `ClientNotifyDeath`
- `MulticastSpawnTracerEffect`

Bad examples:

- `FireWeapon` - Does not indicate its an RPC of some kind.
- `ServerClientBroadcast` - Confusing.
- `AllNotifyDeath` - Use `Multicast`, never `All`.
- `ClientWeapon` - No verb, ambiguous.

3.3.2 All Functions Must Have Return Nodes lint supported

All functions must have return nodes, no exceptions.

Return nodes explicitly note that a function has finished its execution. In a world where blueprints can be filled with `Sequence`, `ForLoopWithBreak`, and backwards reroute nodes, explicit execution flow is important for readability, maintenance, and easier debugging.

The Blueprint compiler is able to follow the flow of execution and will warn you if there is a branch of your code with an unhandled return or bad flow if you use return nodes.

In situations like where a programmer may add a pin to a Sequence node or add logic after a for loop completes but the loop iteration might return early, this can often result in an accidental error in code flow. The warnings the Blueprint compiler will alert everyone of these issues immediately.

3.3.3 No Function Should Have More Than 50 Nodes lint supported

Simply, no function should have more than 50 nodes. Any function this big should be broken down into smaller functions for readability and ease of maintenance.

The following nodes are not counted as they are deemed to not increase function complexity:

- Comment
- Route
- Cast
- Getting a Variable
- Breaking a Struct
- Function Entry
- Self

3.3.4 All Public Functions Should Have A Description lint supported

This rule applies more to public facing or marketplace blueprints, so that others can more easily navigate and consume your blueprint API.

Simply, any function that has an access specifier of Public should have its description filled out.

3.3.5 All Custom Static Plugin `BlueprintCallable` Functions Must Be Categorized By Plugin Name lint unsupported

If your project includes a plugin that defines `static` `BlueprintCallable` functions, they should have their category set to the plugin's name or a subset category of the plugin's name.

For example, `Zed Camera Interface` or `Zed Camera Interface | Image Capturing`.

3.4 Blueprint Graphs lint unsupported

This section covers things that apply to all Blueprint graphs.

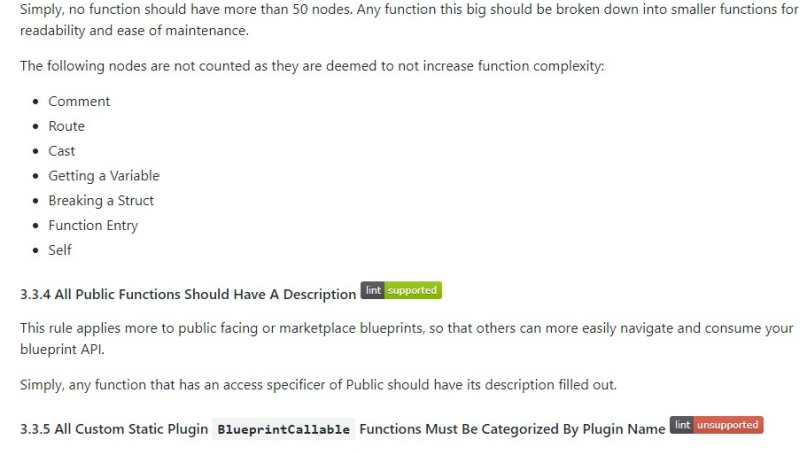
3.4.1 No Spaghetti lint unsupported

Wires should have clear beginnings and ends. You should never have to mentally untangle wires to make sense of a graph. Many of the following sections are dedicated to reducing spaghetti.

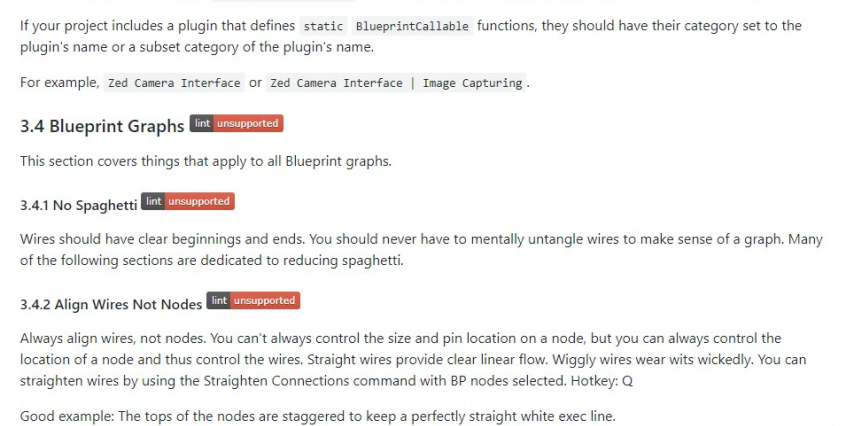
3.4.2 Align Wires Not Nodes lint unsupported

Always align wires, not nodes. You can't always control the size and pin location on a node, but you can always control the location of a node and thus control the wires. Straight wires provide clear linear flow. Wiggly wires wear wits wickedly. You can straighten wires by using the Straighten Connections command with BP nodes selected. Hotkey: Q

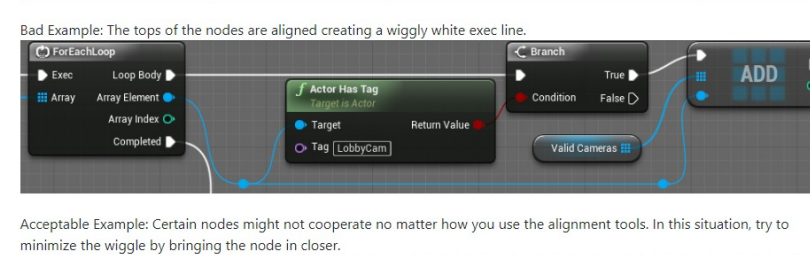
Good example: The tops of the nodes are staggered to keep a perfectly straight white exec line.



Bad Example: The tops of the nodes are aligned creating a wiggly white exec line.



Acceptable Example: Certain nodes might not cooperate no matter how you use the alignment tools. In this situation, try to minimize the wiggle by bringing the node in closer.



3.4.3 White Exec Lines Are Top Priority lint unsupported

If you ever have to decide between straightening a linear white exec line or straightening data lines of some kind, always straighten the white exec line.

3.4.4 Graphs Should Be Reasonably Commented lint unsupported

Blocks of nodes should be wrapped in comments that describe their higher-level behavior. While every function should be well named so that each individual node is easily readable and understandable, groups of nodes contributing to a purpose should have their purpose described in a comment block. If a function does not have many blocks of nodes and its clear that the nodes are serving a direct purpose in the function's goal, then they do not need to be commented as the function name and description should suffice.

3.4.5 Graphs Should Handle Casting Errors Where Appropriate lint unsupported

If a function or event assumes that a cast always succeeds, it should appropriately report a failure in logic if the cast fails. This lets others know why something that is 'supposed to work' doesn't. A function should also attempt a graceful recover after a failed cast if it's known that the reference being casted could ever fail to be casted.

This does not mean every cast node should have its failure handled. In many cases, especially events regarding things like collisions, it is expected that execution flow terminates on a failed cast quietly.

3.4.6 Graphs Should Not Have Any Dangling / Loose / Dead Nodes lint unsupported

All nodes in all blueprint graphs must have a purpose. You should not leave dangling blueprint nodes around that have no purpose or are not executed.

Back to Top

4. Static Meshes lint partial support

This section will focus on Static Mesh assets and their internals.

Sections

- 4.1 UVs
- 4.2 LODs
- 4.3 Modular Socketless Snapping
- 4.4 Must Have Collision
- 4.5 Correct Scale

4.1 Static Mesh UVs lint supported

If Linter is reporting bad UVs and you can't seem to track it down, open the resulting `.log` file in your project's `Saved/Logs` folder for exact details as to why it's failing. I am hoping to include these messages in the Lint report in the future.

4.1.1 All Meshes Must Have UVs lint supported

Pretty simple. All meshes, regardless how they are to be used, should not be missing UVs.

4.1.2 All Meshes Must Not Have Overlapping UVs for Lightmaps lint supported

Pretty simple. All meshes, regardless how they are to be used, should have valid non-overlapping UVs.

4.2 LODs Should Be Set Up Correctly lint unsupported

This is a subjective check on a per-project basis, but as a general rule any mesh that can be seen at varying distances should have proper LODs.

4.3 Modular Socketless Assets Should Snap To The Grid Cleanly lint unsupported

This is a subjective check on a per-asset basis, however any modular socketless assets should snap together cleanly based on the project's grid settings.

It is up to the project whether to snap based on a power of 2 grid or on a base 10 grid. However if you are authoring modular socketless assets for the marketplace, Epic's requirement is that they snap cleanly when the grid is set to 10 units or bigger.

4.4 All Meshes Must Have Collision lint unsupported

Regardless of whether an asset is going to be used for collision in a level, all meshes should have proper collision defined. This helps the engine with things such as bounds calculations, occlusion, and lighting. Collision should also be well-formed to the asset.

4.5 All Meshes Should Be Scaled Correctly lint unsupported

This is a subjective check on a per-project basis, however all assets should be scaled correctly to their project. Level designers or blueprint authors should not have to tweak the scale of meshes to get them to confirm in the editor. Scaling meshes in the engine should be treated as a scale override, not a scale correction.

Back to Top

5. Particle Systems lint supported

This section will focus on Particle System assets and their internals.

Sections

- 5.1 Emitter Naming

5.1 Emitter Naming lint supported

All emitters in a Particle System should be named something descriptive and not left to their default name "Particle Emitter".

Back to Top

6. Levels / Maps lint partial support

See Terminology Note regarding "levels" vs "maps".

This section will focus on Level assets and their internals.

Sections

- 6.1 No Errors Or Warnings
- 6.2 Lighting Should Be Built
- 6.3 No Player Visible Z Fighting
- 6.4 Marketplace Specific Rules

6.1 No Errors Or Warnings lint partial support

All levels should load with zero errors or warnings. If a level loads with any errors or warnings, they should be fixed immediately to prevent cascading issues.

You can run a map check on an open level in the editor by using the console command "map check".

Please note: Linter is even more strict on this than the editor is currently, and will catch load errors that the editor will resolve on its own.

6.2 Lighting Should Be Built lint unsupported

It is normal during development for levels to occasionally not have lighting built. When doing a test/internal/shipping build or any build that is to be distributed however, lighting should always be built.

6.3 No Player Visible Z Fighting lint unsupported

Levels should not have any z-fighting in all areas visible to the player.

6.4 Marketplace Specific Rules lint unsupported

If a project is to be sold on the UE4 Marketplace, it must follow these rules.

6.4.1 Overview Level lint unsupported

If your project contains assets that should be visualized or demoed, you must have a map within your project that contains the name "Overview".

This overview map, if it is visualizing assets, should be set up according to Epic's guidelines.

For example, `InteractionComponent_Overview`.

6.4.2 Demo Level lint unsupported

If your project contains assets that should be demoed or come with some sort of tutorial, you must have a map within your project that contains the name "Demo". This level should also contain documentation within it in some form that illustrates how to use your project. See Epic's Content Examples project for good examples on how to do this.

If your project is a gameplay mechanic or other form of system as opposed to an art pack, this can be the same as your "Overview" map.

For example, `InteractionComponent_Overview_Demo`, `ExplosionKit_Demo`.

Back to Top

7. Textures lint unsupported

This section will focus on Texture assets and their internals.

Sections

- 7.1 Dimensions Are Powers of 2
- 7.2 Texture Density Should Be Uniform
- 7.3 Textures Should Be No Bigger than 8192
- 7.4 Correct Texture Groups

7.1 Dimensions Are Powers of 2 lint unsupported

All textures, except for UI textures, must have its dimensions in multiples of powers of 2. Textures do not have to be square.

For example, `128x512`, `1024x1024`, `2048x1024`, `1024x2048`, `1x512`.

7.2 Texture Density Should Be Uniform lint unsupported

All textures should be of a size appropriate for their standard use case. Appropriate texture density varies from project to project, but all textures within that project should have a consistent density.

For example, if a project's texture density is 8 pixel per 1 unit, a texture that is meant to be applied to a 100x100 unit cube should be 1024x1024, as that is the closest power of 2 that matches the project's texture density.

7.3 Textures Should Be No Bigger than 8192 lint unsupported

No texture should have a dimension that exceeds 8192 in size, unless you have a very explicit reason to do so. Often, using a texture this big is simply just a waste of resources.

7.4 Textures Should Be Grouped Correctly lint unsupported

Every texture has a Texture Group property used for LODing, and this should be set correctly based on its use. For example, all UI textures should belong in the UI texture group.

Back to Top

Contributors

- Michael Allar: GitHub, Twitter
- CosmoMyzrailGorynych
- billymcguffin
- akenatsu

License

Copyright (c) 2016 Gamemakin LLC

See LICENSE

Back to Top

Amendments

We encourage you to fork this guide and change the rules to fit your team's style guide. Below, you may list some amendments to the style guide. This allows you to periodically update your style guide without having to deal with merge conflicts.

};