A concise guide to almost all the C# you'll ever need to know code-wise to make a PS4 game in Unity

Alex Rose, @AlexRoseGames

This guide is to teach yourself. I'd advise not watching youtube tutorials where someone spends 20 minutes showing how to use the hierarchy, just fiddle with stuff and drag stuff around until it works. Add components. Fiddle with classes. If you want to do something, start making a script for it, do the bit you can do and when you're stuck, google it. Realistically, there won't be any tutorials later for your hard, specific problems later so it's good practice not to use them at the beginning.

This document contains all the variables, functions and other shit you'll need to know to make a 2D PS4 game, minus like some very small things that don't matter. The document's reasonably small because there's not actually that much stuff. If you just learn every line of this (there's a reason it's so small – because it all matters) and you know how to tell a machine to do stuff, you can make a PS4 game. You need to know how to work the engine too and the nuances but this covers all the code almost. By the time you can use all of this stuff there won't be much extra stuff you'd need to learn. Knowing maths helps a lot, especially trig (for code based animation e.g.). If you read something and you don't fully get it, google that thing or experiment with it until you do.

First up, get some good version of Visual Studio and use intellisense first of all, it will speed up everything about 9000x. Legitimately, it not only speeds up typing probably 5-10x but it speeds up learning like 10,000 times because it shows you all your variables. Link it TO Unity.

I'd advise using C# because js isn't as powerful. It starts very slightly more difficult but then it's way better afterwards. I don't even know what Booscript is, Unity also supports this, I would advise not using it considering it's like.. called Boo. And no one uses it. If you don't know enough c#, codecademy is a good way of learning.

Non-specific Variables you'll need to use:

void (the vast majority of your functions will return void)

int (integers, whole numbers)

float (we don't use doubles in c#. Because of that, when you write 3.453 you have to write 3.453f, not 3.453 everywhere in your code)

string (we won't use chars but understand that chars use ' ' and strings use " ", and a string is an array of chars)

bool (true or false. NOT EQUIVALENT TO 1 or 0 in c#)

arrays (blah.Length gets the length)

C# Lists (System.Collections.Generic. These are good for arrays when you don't know the original size) (blah.Count gets the length)

Vector2 (just an x and y thing)

Vector3 (x, y, z, which you can cast to Vector2 by writing (Vector2)[my vector3]). These also contain .normalised and .magnitude, which are super useful. If you just want to change x, tough shit, you have to do:

mynewVector = new Vector3(newX, myoldvector.y, myoldvector.z);

Quaternion (don't try to understand it, just use its functions. It's a four dimensional rotation tensor matrix).

Rects (used for gui rectangle shit mainly. Might not need to know this for a long time)

new (e.g. somelist = new List<blah>(); or somearray = new int[5]{1,2,3,4,5};) (You need this for arrays, colors, color32s, vector2s, vector3s, rects, lists and probably other stuff I forgot)

return (give back a variable at the end of a function. Also in voids you can just say return; to kill it straight off)

yield return new (you need this for ienumerators. You can stick a function on the end of there and it won't stop until it's finished doing it. E.g. WaitForSeconds() or Instantiate() if you're spawning a whole scene or something).

Maths shit:

UnityEngine.Random.Range(lowerinclusive,higheruninclusive)

Mathf.PI 3.141592etc.

Mathf.RoundToInt -> converts floats to ints

Mathf.FloorToInt -> does it down

Mathf.CeilToInt -> does it up

Mathf.Rad2Deg -> converts from radians to degrees for rotation shit

Mathf.Deg2Rad

System.Int32.Parse(string) -> turns string into number. Tryparse also exists but fuck try, just get it right.

Mathf.Abs for absolute values.

Mathf.Sin, Mathf.Cos, Mathf.Tan for trig. Mathf.Asin, Mathf.Acos, Mathf.Atan2 for antitrig. Use Atan2(y,x) not Mathf.Atan or it won't work how you want it to in 99% of cases. Just trust me there.

Mathf.Pow is really good for squared trig functions (generally x*x is faster than Mathf.pow(x,2) though but it saves you writing it on two lines or doing the calculation twice) and gravity.

Mathf.Sqrt obvs too.

Terms:

public (interactable from other classes and you can change it in the editor). You're going to be using this a lot.

static (The same for every class, can't change in the editor). You'll use this when you're better at knowing shit. It's handy to make a public static version of everything there's one of like your camera and your player, so you can easily call them from anywhere.

// Anything after this is a comment and won't compile

/* everything here is commented */

Statements:

if

else if

else

for(int i=0;i<blah.length;i++){}

foreach (e.g. foreach(Enemy groar in FindObjectsOfType(typeof(Enemy)) as Enemy[]){ Destroy(blah.gameObject) } will destroy all groars. Easier example, foreach(Coin coin in coins [some List<coin> you made before]){ coin.ping(); } or whatever.

switch (if you want), as in switch(somevariable) case 1: [whatever happens if somevariable is 1] break; case 2: [somevariable == 2], case

you can use while but.. most of the time there's a better way

operators (if you don't know these look them up):

blah++, blah--, blah*=, blah+=, blah-=

== (you can test for blah==null too for gameobjects you haven't defined yet). There's none of this === javascript bullshit here.

!=

>, >=, <, <=

c# tricks:

somebool = !somebool; inverts a bool

myVariable = (facingRight ? 1: 0); is a really easy way of writing if(facingRight==true) myVariable = 1; else myVariable = 0;

Unity variables you need to use:

isStatic -> this is in the top right of the editor. Tick this shit if the thing never moves, makes it way more efficient. NEVER TICK IT OTHERWISE.

Tag is a group something's tagged as. I personally avoid tags except for things like.. if they kill you. But you can easily do this by checking for a non-null class anyway.

Layer is for collision layers

enabled -> if this component is running.

Transform – this is the core of all objects. It has their spatial dimensions. It contains (**ALL REALLY IMPORTANT**):

transform.position -> position in world space

transform.rotation -> rotation in world space, this is a Quaternion. More on that later.

transform.parent = the thing it's attached to

transform.localPosition = its position relative to its parent

transform.localRotation

transform.localScale (YOU HAVE TO MANIPULATE THIS, YOU CAN'T CHANGE REAL SCALE)

you can GET transform.lossyScale but it's mostly useless

transform.GetChild(0) will get the first child. Also you can get all children of "bloop" with:

foreach(Transform blah in bloop.transform)

You can move shit around by its transform.position/scale and rotation where physics doesn't matter like aesthetic stuff, background stuff and stuff the player can't touch, but if it's something you collide with, never do that shit or it will break everything. Use rigidbodies.

GameObject (you can grab this gameobject component with blah.gameObject to fiddle with it). This is what you'll mostly deal with, spawn, destroy, grab components off.

Collider/Collider2D (this is how things with rigidbodies/rigidbody2ds hit each other. It has a thing called isTrigger. Triggers are great for starting events where it doesn't stop the player, like making someone talk when you walk in a room or something. Colliders also send collision messages so you know what's up. More on that later)

Rigidbody/Rigidbody2D (do all your physics through here. They contain mass, drag, angularvelocity, angulardrag, gravityScale etc. and you should do movement through here by doing Rigidbody2D.velocity += blah or using AddForce). isKinematic says whether it collides with stuff or whether it just moves on its own. Other stuff will still collide with kinematic objects.

Renderer/SpriteRenderer -> enable/disable this to make it appear/disappear. This also has "material" on it and "sharedMaterial". If you fuck with a sharedMaterial it will change forever in the editor but change all objects. Changing one Material will change draw calls.

Materials also have the option to setfloats, setcolors etc. which is useful when you're writing shaders, but you probably aren't right now so figure that out when you need to.

Color/Color32, color goes from 0-1, color32 uses bytes from 1-255. You can do maths with bytes but you'll have to cast to byte by writing (byte)[some int]. That's good for fading shit out. And use Mathf.RoundToInt first. If you want to grab components you do somecolor.r, somecolor.g, somecolor.b, somecolor.a and it will return 0-1, so you can use that to find like.. if you're faded.

Also shit like AudioSource/Camera/Light, you'll learn as time goes on (e.g light distance, intensity). Camera.main is important, that's your main camera obvs. You can get that to work in the tags.

Unity Functions you need to use:

Awake() – Runs at the very, very beginning. Use this scarcely, for things that NEED to happen before anything else, like choosing static variables. E.g. in RudeBear the first thing that happens is "Bear = this" where Bear is a public static variable, so any other script can say "RudeBear.Bear.Die();" or whatever

Start() – This is where you set your variables. You HAVE to. Even if you put int blah=5; if that ever changes when you reload the scene it'll still be changed. You need to set all your variables here and create your arrays and shit.

Update() – Runs every frame. You can handle input here (although I don't). e.g. it'll immediately know if you've clicked. But if when you click, something happens, think about doing it in FixedUpdate or you game will become framerate dependent. If I apply some force in Update() then

the game will change if you're at 30fps or 60fps. You can divide by Time.deltaTime but that's what morons do, don't listen to that.

FixedUpdate() – This runs every 0.02 seconds by default. This is where all physics happens. Always control gameplay through here like physics.

OnDestroy() – When something gets destroyed. I'd really recommend not using this except for memory stuff and making your own function when you destroy stuff, because ondestroy changes stuff in the editor

OnTriggerEnter/OnTriggerEnter2D(Collider/Collider2D Hit)

These will let you see if a trigger's been triggered. Normally in Start you'll set some "bool triggered;" to false, then set it to true in here if it reaches the right criteria (like in my case Hit.gameObject == RudeBear.Bear.gameObject when RB hits something)

OnTriggerStay/OnTriggerExit and 2D versions also exist and work the same. Stay is every fixedupdate when you're on it.

OnCollisionEnter/OnCollisionEnter2D(Collision/Collision2D Hit). These are great for detecting collisions. You can get really good information out of here like **SUPER EXAMPLE FOR DETECTING IF YOUR DUDE HIT THE FLOOR:**

```
void OnCollisionEnter2D(Collision2D Hit){

	foreach(ContactPoint2D contact in Hit.contacts){

[[for every part of your object that touched the other object, like the corners or the bottom or whatever]]

		if(contact.normal.y>0.8f){ [[if the y normal of the point (-1 to 1) is above 0.8, as in.. up]]

			grounded = true; [[the player hit the floor]];

			break; [[quit the loop, I got what I came for]]

		}

	}

}
```

Easier still if you're dealing with squares that stay upright you can always just do if(hit.contacts[0].normal.y>0.8f) blah();

Again, OnCollisionStay/Exit exist and in 2D.

IgnoreLayerCollision ignores collisions between 2 layers. Useful.

Fuck OnEnable, do it better.

Unity Functions to call:

Instantiate(GameObject, position, rotation) [[Instantiate creates a clone of a prefab. A prefab is just any gameobject you drag into your work folder and then spawn in your game]]

or: GameObject blah = Instantiate(GameObject, position, rotation) as GameObject; [[that way you can fiddle with it after you spawn it]]

Destroy(object,time) destroys after time. You can't destroy transforms. If you Destroy(audioSource) or something, it will destroy the audio component, not the object. If you want to destroy the object use blah.gameObject, but a lot of the time you might just want to disable their colliders and stuff.

GetComponent<someComponent>() grabs a class off anything. REALLY FUCKING USEFUL.

Application.loadLevel(blah) can load any levels in the build by string name or int id.

GameObject.FindGameObjectsWithTag("sometag") will create an array of tags. DON'T DO THIS EVERY FRAME, DO IT ONE TIME AND HOLD THE ARRAY.

Likewise FindObjectsOfType(typeof(someClass)) as someClass[] will find all objects that have someClass on them and make an array

gameObject.SendMessage("blah","bloop") will activate function blah in all classes that have a function called blah on that gameobject with parameter bloop. Don't touch this shit unless you're doing it en masse or something.

Raycast2D and Raycast. This is how you detect clicking on stuff. Don't use it for physics, if someone recommends that they're an idiot. Look up RaycastHit as well. Figure it out once and copy and paste everywhere. Uses layer mask as well which uses bitwise operations, just know that ~18 will avoid layer 18, ~18&~12 will avoid both. Only use "nots" everywhere and combine them, don't use positives.

Input shit:

Input.GetKeyDown(Keycode.A) gets if the letter A was hit down. Use in input. You can use Input.GetKey to see if it's currently down, Input.GetButton/GetButtonUp/GetButtonDown is for your defined axes.

Input.GetAxis works for axes you define in the inputmanager. This is great for controllers, also works for keyboard buttons.

Input.GetAxisRaw does the same thing but without bullshit. In Input.GetAxis if you go from the stick being 100% down to 0% down in a split second, it will gradually change the value from 100 to 0. You probably want Raw because you want full control.

There's also like Input.GetMousePosition or something that gives you that. You can get it as a percentage of the screen by dividing by Screen.width and Screen.height in a new Vector2

Input.GetMouseButton(0 or 1 for left/right). There's also getmousebuttondown and up. This will probs involve raycasting

There's also Touches you can use but fuck mobile amirite. It's easy anyway, it's just a foreach loop through all touches.

HOW TO MAKE SHIT MOVE WHERE YOU WANT IT TO MOVE:

If physics doesn't matter:

Move instantly:

transform.position = new Vector3(x,y,z);

transform.rotation = Quaternion.Euler(x,y,z) in degrees. It complains it's deprecated but if you use Quaternion.EulerAngles(x,y,z) it's in radians because fuck degrees.

There's also Quaternion.LookRotation to look at another object and transform.rotation.Rotate, which rotates every step. You should know Vector3.forward/back etc.

Also Quaternion.LookAt looks in a particular direction, but it's not smooth so I prefer to use lookrotation with slerp.

Move 1/10th of the way to the desired value every fixedupdate (put this in FixedUpdate()). This will make it look great.

transform.position += (finalposition-transform.position)*0.1f;

This is a great magic formula (currentval +=(finalval-currentval)*speedtodoitat).

transform.rotation = Quaternion.Slerp(transform.rotation,finalrotation,0.1f); (this time 0.1f it's time period. This will do it smoothly).

How to do it with physics: (THIS IS IMPORTANT IF YOU WANT STUFF TO COLLIDER PROPERLY)

Instantly move: DON'T INSTANTLY MOVE IT IF YOU WANT PHYSICS.

Move smoothly:

Rigidbody2D.velocity += (desiredposition-transform.position)*0.1f/Time.fixedDeltaTime;

Will get there at the right point. You can even just set a desiredposition that you vary with trigonometry or.. sometimes I use an “effectivedesiredposition” and then add some trig onto it to make my fairy fly up and down at the point it wants to be at.

PRACTICALLY instantly move:

If you do it as *1f it will do it instantly, but because of orders of errors it will end up wiggling round the point you want. You can fix that using drag or by adding another (velocity*=0.6f) every turn to stop that from happening if you’re using velocities/angvels, so:

```
Rigidbody2D.velocity += (desiredposition-transform.position) /Time.fixedDeltaTime;

Rigidbody2D.velocity *= 0.6f;
```

Good for kinematic shit, for non kinematic just use drag.

If you want to move 2d rotations by angularvelocity/torque, tweet me and I’ll tell you how to not make the angles fuck up.

How IEnumerators work (TIMED EVENTS):

When I want to call one I say: StartCoroutine(myfunction()); Then elsewhere it says

```
IEnumerator myfunction(){
        yield return new WaitForSeconds(1f);
        nowdostuff();
}
```

This runs in parallel to all my other code, so if I say

```
i=0;
i++;
StartCoroutine(myfunction());
i++;
```

i will be 2 by the time myfunction() ends. So don’t fiddle with shit that you’re using in your coroutine in the same time. Hell, avoid coroutines where you can, use counter ints in FixedUpdate, but coroutines can work well for like.. dialogue and shit. Just try not to put your core game logic in there.

StopAllCoroutines() will kill all of them but watch yo’self.

Magic awesome advanced shit

[ExecuteInEditMode] -> lets you do stuff in the editor from Start() OnGizmoDraw or whatever it’s called and Update(). Use with if(!Application.isPlaying) and #if UNITY_EDITOR #endif, which is a

preprecessor directive that means the bit in between those tags won't compile. You don't want this stuff running while your game is on.

DestroyImmediate() is good for that kind of shit in the editor.

One more example, making screen fade to black before loading a new level (you can also use Application.loadLevelAsync to do this more efficiently but I'm assuming you don't have unity pro right now. If you do, figure that out, it's not hard. You just declare it and instantly tell it it's not allowed to load, then when you're done tell it it is).

```
public SpriteRenderer fade;

bool fading;

 [[drag a giant sprite over the entire camera and set the alpha to 0 so it's not visible and turn it off too on Start()]].

void Start(){

	fade.enabled=false;

	fading=false;

}

void FixedUpdate(){

	if(fading){

		fade.color += new Color32(0,0,0,3); //go up 3 bytes of alpha every 0.02 seconds

		if(fade.color>=0.95f){ //if it's above a certain amount

			fade.color = new Color32(0,0,0,255); // go pure black, it pretty much is anyway

			fading=false;

			Application.loadLevel(1);

	}

}
```

Etc.

There’s probably more stuff I forgot that I’ll add at some later point but that’s the vast majority of the stuff you need to know, and once you can do all this you’ll know enough to just fiddle with stuff until you get there anyway.

If you want help with particle systems, tweet me. If you don’t know maths don’t even bother trying to code with them, just use the editor settings. If you do, I can probs help you on twitter.