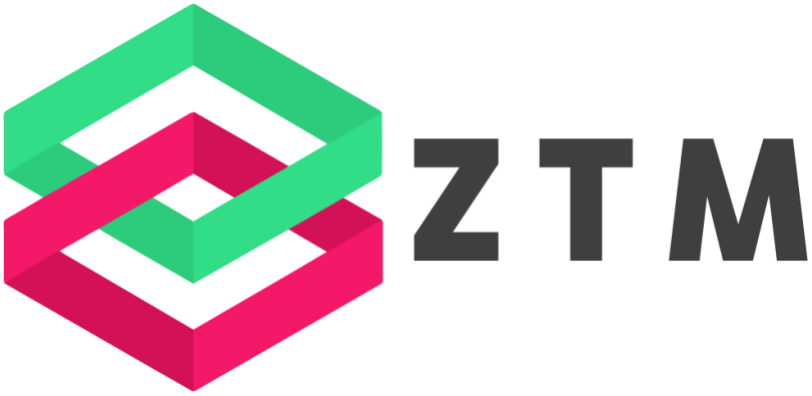

UNITY GAME DEVELOPMENT CHEAT SHEET

LUIS RAMIREZ JR.

V1.01

HEEELLLOOOOO!

I'm Andrei Neagoie, Founder and Lead Instructor of the Zero To Mastery Academy.

After working as a Senior Software Developer over the years, I now dedicate 100% of my time to teaching others valuable software development skills, help them break into the tech industry, and advance their careers.

In only a few years, **over 1,000,000 students** around the world have taken Zero To Mastery courses and many of them are now working at top tier companies like Apple, Google, Amazon, Tesla, IBM, Facebook, and Shopify, just to name a few.

This cheat sheet, created by our Unity 3D Game Development instructor (Luis Ramirez Jr.) provides you with the key Unity concepts that you need to know and remember.

If you want to not only learn Unity but also get the exact steps to build your own games and get hired as a Game Developer, then check out our Career Paths.

Happy Coding!
Andrei

Founder & Lead Instructor, Zero To Mastery
Andrei Neagoie

Unity Cheatsheet

Table of Contents

Keyboard Shortcuts

Boost your productivity by remembering the most common keyboard shortcuts in the Unity Editor.

Action	Key Binding
View Tool	*Q*
Move Tool	*W*
Rotate Tool	*E*
Scale Tool	*R*
Rect Tool	*T*
Transform Tool	*Y*
New Game Object	Windows: *CTRL* + *SHIFT* + *N* Mac: *CMD* + *SHIFT* + *N*
New Child Game Object	*ALT* + *SHIFT* + *N*
Toggle Window Size	*SHIFT* + *SPACE*
Duplicate	Windows: *CTRL* + *D* Mac: *CMD* +*D*
Play/Pause	Windows: *CTRL* + *P* Mac: *CMD* + *P*
Focus Game Object	*F*

If you're using Visual Studio Code, here are some handy keyboard shortcuts.

Action	Key Binding
Command Palette	Windows: *CTRL* + *SHIFT* + *P* Mac: *CMD* + *SHIFT* + *P*
Quick File Open	Windows: *CTRL* + *P* Mac: *CMD* + *P*
Toggle Sidebar	Windows: *CTRL* + *B* Mac: *CMD* + *B*
Multi-Select Cursor	Windows: *CTRL* + *D* Mac: *CMD* + *D*
Copy Line	Windows: *SHIFT* + *ALT* + *UP* or *SHIFT* + *ALT* + *DOWN* Mac: *OPT* + *SHIFT* + *UP* or *OPT* + *SHIFT* + *DOWN*
Comment Code Block	Windows: *SHIFT* + *ALT* + *A* (Multi-line comment), *CTRL* + *K* + *C* (Single-line comment) Mac: *SHIFT* + *OPT* + *A*
Show All Symbols	Windows: *CTRL* + *T* Mac: *CMD* + *T*
Trigger suggestion and Trigger parameter hints	Windows: *Ctrl* + *Space*, *Ctrl* + *Shift* + *Space* Mac: *CMD* + *Space*, *CMD* + *Shift* + *Space*

Show References	Windows: *Shift* + *F12* Mac: *Shift* + *F12*
Global Find	Windows: *CTRL* + *SHIFT* + *F* Mac: *CMD* + *SHIFT* + *F*

MonoBehaviour

The `MonoBehaviour` class is defined by Unity to help you transform your class into a component for game objects:

```
using UnityEngine;

public class ExampleClass : MonoBehaviour
{

}
```

Script Lifecycle

All components that derive from the `MonoBehaviour` class executes a number of event functions in a predetermined order. They're the following:

- `Awake()` - This function is called after a game object has been instantiated.
- `OnEnable()` - This function is called when a game object is enabled.
- `Start()` - This function is called before the first frame update.
- `Update()` - This function is called on every frame.
- `LateUpdate()` - This function is called on every frame **after** the `Update()` function is called.
- `OnBecameVisible()` - This function is called when the current game object becomes visible via any camera.
- `OnBecameInvisible()` - This function is called when the current game object no longer becomes visible via any camera.
- `OnDrawGizmos()` - This function is called for drawing gizmos in the **scene** window.
- `OnGUI()` - This function is called multiple times for GUI events.
- `OnApplicationPause()` - This function is called when the game is paused via the Unity editor.

- `OnDisable()` - This function is called when the game object is disabled.
- `OnDestroy()` - This function is called when the game object is destroyed.

There's a lifecycle function called `FixedUpdate`, which is called a fixed number of times per second. You can configure the frequency in the **Edit ▸ Project Settings ▸ Time ▸ Fixed Timestep**.

Serializing Variables

Unity is capable of performing serialization on your variables. Serialization is the process of transforming your data into a format that can be read and edited from the Unity editor. Variables can be serialized based on either the attribute or access modifiers applied to a variable.

If a variable is `public`, it'll be automatically serialized:

```
public int age = 10;
```

If you don't want public variables to be serialized, you can use the `NonSerialized` attribute from the `System` namespace like so:

```
[NonSerialized] public int age = 10;
```

Private variables not serialized. However, if you want them to be serialized, Unity has an attribute called `SerializeField` that can be found from the `Unity` namespace. It can be applied like so:

```
[SerializeField] private int age = 10;
```

Instantiating Game Objects

New game objects can be inserted into the scene programmatically by calling the `Instantiate()` function. This function has three arguments.

- The Game Object
- (Optional) Global Position

- (Optional) Rotation

```
Instantiate(someGameObject);
Instantiate(someGameObject, new Vector3(0, 0, 10));
Instantiate(someGameObject, new Vector3(0, 0, 10), Quaternion.identity);
```

Finding Game Objects

The `GameObject` class has a few methods for finding game objects within the hierarchy.

- `Find()` - Searches for a game object based on its name.
- `FindGameObjectsWithTag()` - Finds multiple game objects that are stored in an array and returned. Uses a game object's tag to find them.
- `FindWithTag()` - Finds a game object based on its tag.

```
GameObject myObj = GameObject.Find("NAME IN HIERARCHY");
GameObject myObj = GameObject.FindGameObjectsWithTag("TAG");
GameObject myObj = GameObject.FindWithTag("TAG");
```

Destroying Game Objects

Game objects can be destroyed by calling the `Destroy()` function:

```
Destroy(gameObject);
```

Components

By themselves, game objects can't do anything by themselves. We must add components to add specific functionality. If you add custom components, you might want to select other components. Unity offers various solutions to grab components from a game object.

The first solution is to use the `GetComponent()` function. There are different ways to call this function. The most common is to pass in the name of the class for the component as a generic like so:

```
AudioSource audioSource = GetComponent<AudioSource>();
```

Alternatively, you can use the `typeof` keyword and assert the value like so:

```
AudioSource audioSource = GetComponent(typeof(AudioSource)) as AudioSource;
```

Lastly, you can pass in the name of the component as a string like so:

```
AudioSource audioSource = GetComponent("AudioSource") as AudioSource;
```

Vectors

Vectors are used for the positions of game objects. You can use them to help you calculate distance and movement. There are two vector classes, which are `Vector2` and `Vector3`.

`Vector2` is primarily used for 2D games that have two coordinates (x, y). `Vector3` is primarily used for 3D games that have three coordinates (x, y, z).

X = Left/Right Y = Up/Down Z = Forward/Back

Unity supplies you with constants that can be accessed from their respective vector class.

Vector 3

```
Vector3.right   /* (1, 0, 0)  */
Vector3.left    /* (-1, 0, 0) */
Vector3.up      /* (0, 1, 0)  */
Vector3.down    /* (0, -1, 0) */
Vector3.forward /* (0, 0, 1)  */
Vector3.back    /* (0, 0, -1) */
Vector3.zero    /* (0, 0, 0)  */
Vector3.one     /* (1, 1, 1)  */
```

Vector2

```
Vector2.right /* (1, 0)  */
Vector2.left  /* (-1, 0) */
Vector2.up    /* (0, 1)  */
Vector2.down  /* (0, -1) */
Vector2.zero  /* (0, 0)  */
Vector2.one   /* (1, 1)  */
```

If you're just interested the direction of a specific vector, you can access its `normalized` property like so:

```
myVector.normalized
```

The distance between two vectors can be calculated with the `Vector3.Distance()` method, which returns a `float`:

```
float distance = Vector3.Distance(vectorOne, vectorTwo);
```

Quaternion

Quaternions are used for the rotation of the game object. The current game object's rotation can be read/updated from via the `transform.rotation` property.

Unity has a function called `Quaternion.LookRotation()`. This function returns a Quaternion that takes in a vector. Typically, the vector would be the position of the game object you would like to look at.

```
Quaternion.LookRotation(gameObjectPosition);
```

Physics Events

If you add a collider component to a game object, you can detect collision events from your components by defining a specific set of methods. The following methods are only called when you have a **Collider** or **Rigid Body** component on both game objects.

- `OnCollisionEnter` - This function is called once when another object has collided with the current game object.
- `OnCollisionStay` - This function is called on every frame when another object has collided with the current game object.
- `OnCollisionExit` - This function is called once when an object exits the collision zone of the current object.

```
private void OnCollisionEnter(Collision hit) {
  Debug.Log($"{gameObject.name} hits {hit.gameObject.name}");
}
private void OnCollisionStay(Collision hit) {
  Debug.Log($"{gameObject.name} is hitting {hit.gameObject.name}");
}
private void OnCollisionExit(Collision hit) {
  Debug.Log($"{gameObject.name} stopped hitting {hit.gameObject.name}");
}
```

The following functions are only called when the **On Trigger** option is turned on from the respective collider component.

- `OnTriggerEnter` - This function is called when another object has collided with the current game object.
- `OnTriggerStay` - This function is called on every frame when another object has collided with the current game object.
- `OnTriggerExit` - This function is called once when an object exits the collision zone of the current object.

```
private void OnTriggerEnter(Collider hit) {
  Debug.Log($"{gameObject.name} hits {hit.gameObject.name}");
}
private void OnTriggerStay(Collider hit) {
  Debug.Log($"{gameObject.name} is hitting {hit.gameObject.name}");
}
private void OnTriggerExit(Collider hit) {
  Debug.Log($"{gameObject.name} stopped hitting {hit.gameObject.name}");
}
```

Lastly, there are counterpart functions for 2D colliders. The functions share the same name as the 3D functions, but have the word `2D` appended at the end. The same goes

for the parameter's type. It's `Collision2D` instead of `Collision`.

- `OnCollisionEnter2D`
- `OnCollisionStay2D`
- `OnCollisionExit2D`
- `OnTriggerEnter2D`
- `OnTriggerStay2D`
- `OnTriggerExit2D`

```
private void OnCollisionEnter2D(Collision2D hit) {
  Debug.Log($"{gameObject.name} hits {hit.gameObject.name}");
}
private void OnCollisionStay2D(Collision2D hit) {
  Debug.Log($"{gameObject.name} is hitting {hit.gameObject.name}");
}
private void OnCollisionExit2D(Collision2D hit) {
  Debug.Log($"{gameObject.name} stopped hitting {hit.gameObject.name}");
}
private void OnTriggerEnter2D(Collision2D hit) {
  Debug.Log($"{gameObject.name} hits {hit.gameObject.name}");
}
private void OnTriggerStay2D(Collision2D hit) {
  Debug.Log($"{gameObject.name} is hitting {hit.gameObject.name}");
}
private void OnTriggerExit2D(Collision2D hit) {
  Debug.Log($"{gameObject.name} stopped hitting {hit.gameObject.name}");
}
```

Custom Unity Events

Custom events can be created to help you communicate between various game objects. First, you must include the correct namespace:

```
using UnityEngine.Events;
```

Next, you can create a custom event by defining a variable with the `UnityEvent` type like so:

```
public event UnityEvent OnCustomEvent;
```

Unity has another data type called `UnityAction` for creating custom events:

```
public event UnityAction OnCustomEvent;
```

The main difference is that the `UnityEvent` type can be serialized into the Unity editor. If you don't need an event to be publicly modifiable through the Unity editor, you're better off with the `UnityAction` type.

You can use generics to send additional data with your event. Up to 4 data types are supported.

```
// 1 Parameter
public event UnityAction<int> OnCustomEvent;
// 2 Parameters
public event UnityAction<int, float> OnCustomEvent;
// 3 Parameters
public event UnityAction<int, float, bool> OnCustomEvent;
// 4 Parameters
public event UnityAction<int, float, bool, string> OnCustomEvent;
```

Scriptable Objects

Scriptable objects are data containers. They can be a great way to centralize data that can be used across game objects and components. You can create a scriptable object by using the `ScriptableObject` class from the `UnityEngine` namespace.

```
using UnityEngine;

[CreateAssetMenu(fileName = "Data", menuName = "Parent Menu/Child Menu", order = 1)]
public class AudioSO : ScriptableObject
{

}
```

Scriptable objects allow users to create files from these classes. You can add a menu item by using the `CreateAssetMenu` attribute, which has the following parameters:

- `fileName` - The name of the file created by Unity when the option is selected.

- `menuName` - The menu name that will be displayed in the create menu.
- `order` - The priority of the menu item when positioned alongside the other menu items.

Useful Links

- Unity Documentation
- Unity Blog
- Unity Learn
- Unity Asset Store
- Order of execution for event functions