Unity3D Tutorial – Scripting to Shoot a Wandering Entity

0. About This Tutorial

1 This is the third tutorial in a series of Unity3D tutorials for Windows, it expects comfort with the Unity3D user interface, Windows concepts, and at least a very basic understanding of scripting.

2 This tutorial shows how to make a simple shooting game with **C#** with **Unity3D** and **MonoDevelop**. This is a programming exercise! **If you have *never* done any kind of computer programming before, this tutorial may be very challenging for you!** It's a good idea to know a programmer who you can ask questions, or if you have internet access, be ready to ask your favorite search engine. New programmers should remember: *trying things out on your own* is what it takes to become good at programming!

1. Starting Point

1 For this tutorial, create a new project within Unity3D: press ⟨ File ⟩, and select ⟨ New Project... ⟩.

2 ⟨ Browse... ⟩ to the ⟨ Desktop ⟩, and create a ⟨ New folder ⟩ (**Right-Mouse Button** in the *File Chooser* and select ⟨ New ▸ Folder ⟩, or just press **Ctrl + Shift +N**)

3 Name the folder ⟨ unity3d scripted shooting ⟩, ⟨ Select Folder ⟩, then ⟨ Create ⟩.

2. Create a Simple 3D Game World

1 As in the first tutorial, create a very simple 3D game world with a **First Person Controller**:
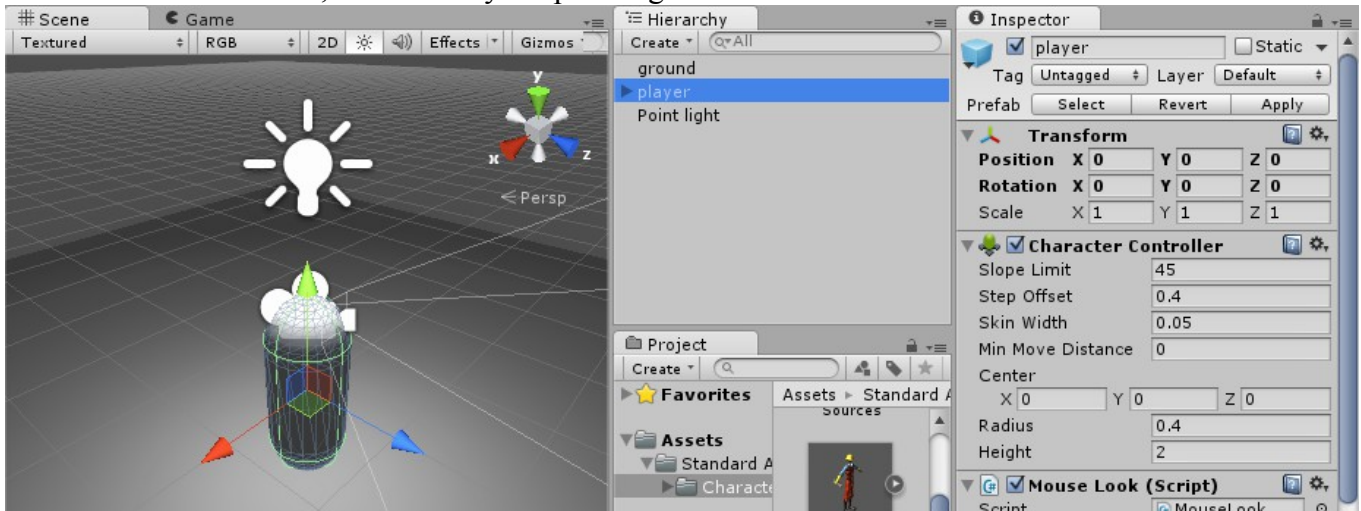
2 **Delete** the **Main Camera** (it will be replaced by the camera on the **First Person Controller**).

3 ⟨ Create ⟩ a **Cube** "ground" (⟨ Hierarchy ⟩ → ⟨ Create ⟩ →**Cube**, **Position**: 0,-2,0 ⟨ X 0 Y -2 Z 0 ⟩, **Scale**: 20,1,20, rename the cube to "ground").

4 Create a **Light** (⟨ Hierarchy ⟩ → ⟨ Create ⟩ →**Point Light**, **Position**: 0,2,0).

5 Create a **First Person Controller** (⟨ Assets ⟩ → ⟨ Import Package ▸ ⟩ → ⟨ Character Controller ⟩ → ⟨ Import ⟩, then from ⟨ Project ⟩, ▼**Assets**, ▼**Standard Assets**, ►**Character Controller**, *drag-and-drop* the **First Person Controller** to ⟨ Hierarchy ⟩, **Position**: 0,0,0, rename "player").

6 **Save** your scene! Name it "CubeShooter".

7 ⟨ ▶ ⟩ *Play Test* your scene. To move the **First Person Controller** (that was renamed to **player**), move the mouse (*Mouselook*), and use the **WASD** keys for movement.

• Note: These steps are covered in much more detail in the first tutorial of this series

3. Create and Test a Simple Script

1 Lets add a script to re-start a player that fell off of the main platform. Press the **Right-Mouse Button** over 📁 **Assets**, in the 📁 **Project**. Then select **Create**, and **C# Script**.

2 Name the script RespawnAfterFall (you can also rename it with the **Right-Mouse Button** menu).

3 **Double-click**, or press **Enter** on the RespawnAfterFall script in the 📁 **Project**. Then, *wait a minute or two*! The script editor, **MonoDevelop**, takes some time to load.

4 Type the following into the RespawnAfterFall.cs file:

```csharp
using UnityEngine;
using System.Collections;

public class RespawnAfterFall : MonoBehaviour
{
        Vector3 startLocation;
        public float lowestAllowedLocation = -20;
        public int numberOfRespawns = 0;
        void Start ()
        {
                startLocation = transform.position;
        }
        public void Respawn()
        {
                transform.position = startLocation;
                numberOfRespawns++;
                print ("respawning "+gameObject);
        }
        void Update ()
        {
                if (transform.position.y < lowestAllowedLocation) {
                        Respawn ();
                }
        }
}
```

• *Note*: with **MonoDevelop** 4, you can auto-format with **Edit → Format → Format Document**. Different code formatting styles can be set at **Project → Apply Policy**. Create your own with **Tools → Custom Policy**.

5 Save the file in **MonoDevelop** (**Ctrl + S**, or select **File**, then 💾 **Save**   Ctrl+S).

6 Switch back to **Unity3D** (**Click** ◁ in the task bar, or press **Alt+Tab**). Read messages in the **Console** (**Click** the 🔴 error message at bottom, or press **Ctrl + Shift + C**) to discover what is wrong with your code, then fix problems in **MonoDevelop**.

• Debugging with others is helpful, but always try to debug by yourself first!
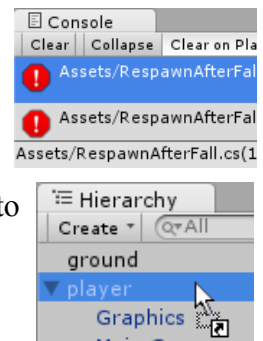
7 If there are no errors, *drag-and-drop* **RespawnAfterFall** from the 📁 **Project** onto **player**. Notice the **Respawn After Fall (Script)** component in **player**.

8 If you accidentally added the script more than once, remove extras by pressing the **Right-Mouse Button** over the extra script component, and select **Remove Component**.

9 ▶ Test the script by running the game, and jumping from the game area. If the game does not bring the player back to the **ground** platform, figure out what the problem is! Ask for help if you need it!

4. About Programming

• *Understanding* code is more important than *typing* code. Take time to read the code, and make sense of it!
• Ask other programmers questions, ask your favorite search engine, predict how code will work and try it!
• The strongest understanding comes from making your own mistakes. You should try things on your own!
• *Understanding* is the price we pay for the power a computer gives us.

5. What the "Simple Script" Is and Does

- These scripts are in **C#** (pronounced "see sharp"). C# is a *C-like Scripting* language. It has a lot in common with the C language, and with other languages like: C++, Java, JavaScript, PHP, Perl, … . C# was developed by Microsoft to be a fast "managed" language using the .NET Framework. Unity3D uses **Mono**, an open-source version of the .NET Framework, with access to all of the C# (.NET version 2.0) libraries.
- Unity3D scripts *inherit* from **MonoBehaviour**, so that they can fit in **GameObject**s as **Component**s.
- To explain specific details about code, programmers often use `//comments`, like those below. The <u>underlined</u> comments below show ideas that are *really* important to understand for Unity3D Game Programming. If you don't understand these yet, look out for more information on these keywords!

```csharp
using UnityEngine;         // allows this code to use standard Unity3D Game Engine features
using System.Collections; // code can use standard C# features from the Collections library

// The class "RespawnAfterFall" inherits from MonoBehaviour in an Object Oriented way.
public class RespawnAfterFall : MonoBehaviour
{
    // Vector3 means "3 Dimensional Vector", this one keeping track of where 'start' is.
    Vector3 startLocation;
    // float means "a positive or negative number variable that can have a decimal part"
    // public means "visible everywhere (including the editor!)"
    public float lowestAllowedLocation = -20;
    // this variable will show (in the Unity editor) how often Respawn() was called
    public int numberOfRespawns = 0;

    // Start is a standard Unity function, called once by an object with this component
    void Start ()
    {
        // When a GameObject with this component starts, it will remember where it is,
        // its transform.position, and store that spot in startLocation.
        // All GameObjects, even "empty" GameObjects, have a Transform called transform.
        startLocation = transform.position;
        print ("respawning "+gameObject); // print some information, helps with testing
    }
    // Respawn is a function (/method/behavior/subroutine/procedure) that can be called by
    // other code (because it is public). "TeleportBackToStart" might be a better name...
    public void Respawn()
    {
        transform.position = startLocation; // compare this line to the 1st in Start()
        numberOfRespawns++; // shorthand for "numberOfRespawns = numberOfRespawns + 1;"
    }
    // Called every frame, based on how fast the game is being drawn
    void Update ()
    {
        // if this object's height location (position on y axis) is too low,
        if (transform.position.y < lowestAllowedLocation)
        {
            Respawn (); // call the Respawn() function.
        }
    }
}
```

- If you are a new programmer, you should know that typing out *your own comments for every line of code* is a great programming exercise! Write comments to explain how you think your code works to *future-you*, who will forget everything you know right now.

6. Bullet Prefab

1 To create a bullet, in `≔ Hierarchy` , `Create ▾` a **Cube**. Name it "bullet",
then press `Add Component` , and select `Physics ▸` , `⚛ Rigidbody`.
**Rigidbody** allows Unity3D to use its built-in physics engine.

- *Rigidbody Physics* means "simplified mechanical physics". *Rigid*-body
assumes that bodies will *not deform* under stress (think crumpled paper).
Simple Rigidbody Physics looks close enough to real physics for games,
without being difficult for a computer to calculate very quickly.

2 Use the values to the right. With the previous **player** values, this will
make the **bullet** show up in front of the **player**, in view of the camera,
just to the right (*Notice*: **Position** 1,0,3 and **Scale** 0.1,0.1,0.1).

3 ▶ Test the game, and notice that the **bullet** falls to the ground.
Uncheck **Use Gravity** ☑, and test again, notice that the bullet does
not fall anymore. *Leave it unchecked*, to simplify the **bullet** for later.
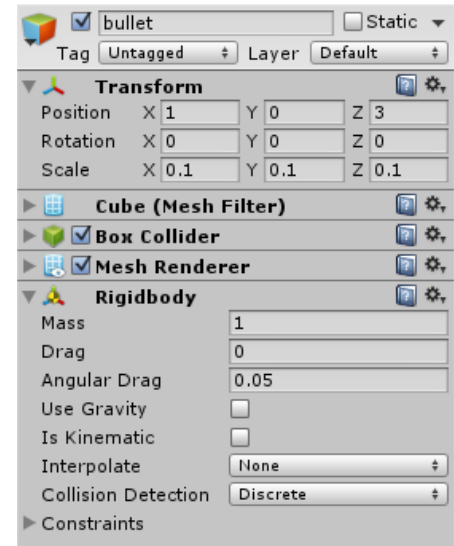
4 Create another **C# Script** like the one created earlier, except call it "InitialVelocity", and use this code:

```csharp
using UnityEngine;
using System.Collections;

[RequireComponent (typeof (Rigidbody))]
public class InitialVelocity : MonoBehaviour
{
        public Vector3 initialVelocity;
        void Start ()
        {
                if (rigidbody.velocity == Vector3.zero)
                {
                        rigidbody.velocity = initialVelocity;
                }
        }
}
```
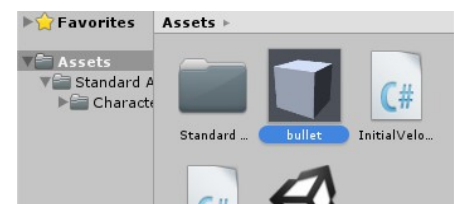
- `[RequireComponent (typeof (Rigidbody))]` will add a **RigidBody** automatically if one isn't there already.

5 Once the script is written and has no errors, attach it to the **bullet** by
either *drag-and-dropping* it, or `Add Component` , `Scripts ▸` ,
InitialVelocity. Set the **Initial Velocity** value to -1,0,0. ▶ Play-test it!
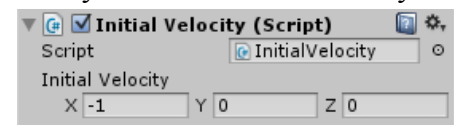
6 To create a bullet that we can easily spawn over and over, we should turn it into a **Prefab** (which stands
for **prefab**ricated object). To do that, *drag-and-drop* the **bullet** from the `≔ Hierarchy` into `📁 Assets` in
`📁 Project` . Now the **bullet** in the game (blue text) is a copy of the **bullet** in `📁 Assets`.

- You can *Drag-and-drop* another **bullet** prefab from the `📁 Project`
to the `≔ Hierarchy` or `# Scene` to make more copies. A prefab
has blue text, and the identical components of the original, including
custom scripts, component values, and even child objects! **Delete** any
extra copies to clean up the scene.
- A prefab can remember **ParticleSystem** components, so feel free to embellish the bullet with sparkles!
- Modifying a prefab copy may "lose the prefab connection". This is fine! Notice the
`Prefab | Select | Revert | Apply ` options at the top of the inspector. To apply any changes back to the
prefab, so that all prefabs are set this way, press **Apply**. Press **Revert** to reset values to match the prefab
source. Press **Select** to find the prefab source in the `📁 Project ` User Interface.

7. Shoot From A Transform

1 Create a new **C# Script**, name it "ShootBullets", and use this code:

```csharp
using UnityEngine;
using System.Collections;

public class ShootBullets : MonoBehaviour
{
        public GameObject bulletPrefab;
        public string shootButton = "Fire1";
        public float bulletInitialSpeed = 10;
        public float bulletDuration = 3;

        void Update ()
        {
                if (Input.GetButtonDown (shootButton))
                {
                        GameObject b = (GameObject)Instantiate(
                                bulletPrefab, transform.position, transform.rotation);
                        InitialVelocity iv = b.GetComponent<InitialVelocity>();
                        iv.initialVelocity = transform.forward * bulletInitialSpeed;
                        Destroy(b, bulletDuration);
                }
        }
}
```

- This script should be on an object that isn't already a collide-able: we don't want bullets to get stuck right when they come out! We need to an **Empty** *child transform* attached to the **player**'s **Main Camera**.
- **Save** before trying this! Setting up a *child transform* is not always intuitive!

2 From `GameObject` , select `Create Empty    Ctrl+Shift+N` , or press **Ctrl + Shift + N**.

3 A new empty **GameObject** was created. Name it "gun".

4 In `Hierarchy` , press ▶ next to **player**, then drag **gun** onto **Main Camera**. (then press ▶ on **Main Camera**). →
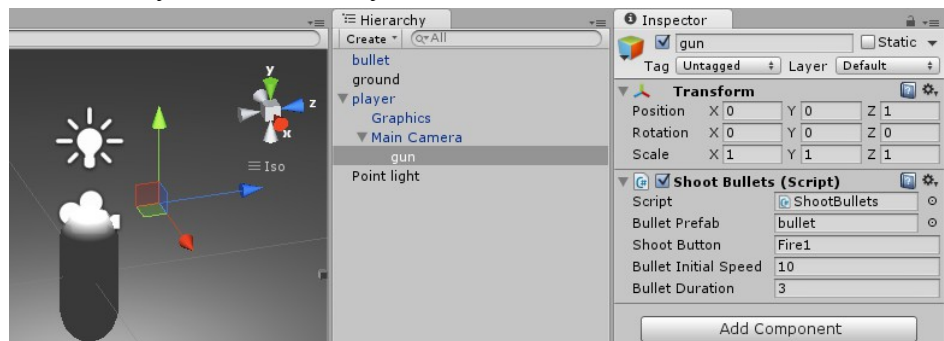
5 Set the **Position** of the **Transform** of gun (a child of **Main Camera**) to 0,0,1.

- The **gun** object is *parented* to the **Main Camera**, just as **Main Camera** is *parented* to player. 0,0,1 is the *local position* of the **gun** object. Wherever **Main Camera** moves, **gun** will always be 0,0,1 away from it.

6 Add **ShootBullets** to the **gun** child object, and select **gun**. *Drag-and-drop* the **bullet** prefab into the **Bullet Prefab** value.
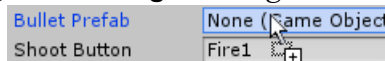
**7** ▶ Play-test! **Left-Mouse Click** or press **Ctrl** to shoot!

- **Input.GetButtonDown** asks the Unity3D **Input** system if a button was just pressed. "Fire1" is a button that the **InputManager** knows about. Find out all of the buttons through `Edit` , `Project Settings ▸` , `Input` . →
- **Instantiate** creates a copy of a known **GameObject**, and places it in the game.
- **GetComponent<***ComponentType***>()** helps **MonoBehaviour**s discover and interact with other components.
- **Destroy** asks the game engine to remove a specific **GameObject** or **MonoBehaviour** from the game.
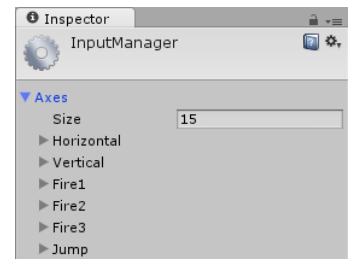
8. Register A Hit

1 `Create ▾` a **Cube** named "targetDummy" to shoot at. Set its **Position** to 0,0,1. Try shooting it in game!

2 Create a new **C# Script**, name it "RegisterHit", and use this code:

```csharp
using UnityEngine;
using System.Collections;

public class RegisterHit : MonoBehaviour
{
        public int hitsBeforeDestroyed = 3;
        LineRenderer hitLine;

        void OnCollisionEnter (Collision collision)
        {
                print (gameObject + " says: Ouch! " + collision.gameObject + " hit me!");
                hitsBeforeDestroyed--;
                if (hitsBeforeDestroyed <= 0) {
                        Destroy(gameObject);
                }
                Vector3 hitSpot = collision.contacts [0].point;
                Vector3 normal = collision.contacts [0].normal;
                hitLine.SetPosition(0, hitSpot - normal);
                hitLine.SetPosition(1, hitSpot);
        }
        void Start()
        {
                GameObject go = new GameObject("hit line");
                go.transform.parent = transform;
                hitLine = go.AddComponent<LineRenderer>();
                hitLine.SetVertexCount(2);
                hitLine.SetWidth(0.1f, 0);
                hitLine.SetPosition(0, Vector3.zero);
                hitLine.SetPosition(1, Vector3.zero);
                hitLine.material = gameObject.renderer.material;
        }
}
```

3 Add **RegisterHit** to **targetDummy,** and ▶ play-test! An arrow draws where it was hit, twice. **targetDummy** disappears being hit 3 times.

- You may notice that if the bullet moves quickly, with **Bullet Initial Speed** of maybe 100, hits don't register! The **bullet** may pass right through! This is called the *Bullet-Through-Paper* problem, discussed at the end of this tutorial. For now, leave the **Bullet Initial Speed** at a reasonable 10.
- **OnCollisionEnter** triggers on a **MonoBehaviour** when the game engine detects new collision.
- **Collision** is information describing how a collision happened.
- **Vector3** is a 3D Vector. Unity3D has an excellent short video introduction to 3D Vectors here: http://unity3d.com/learn/tutorials/modules/scripting/lessons/vector-maths-dot-cross-products
- **LineRenderer hitLine** is just the line-drawing component of a new **GameObject**, used to help draw extra information, to help us understand how things are working. The new **GameObject** is named "hit line", and is *parented* (attached as a child transform) to the object with this **RegisterHit** component.
- **AddComponent<***ComponentType***>()** is used to put a new *ComponentType* onto an object using scripting. This function can even add custom components, like **RegisterHit**, or **RespawnAfterFall**.
- **SetPosition** sets the 3D location of a part of a line being drawn by the **LineRenderer**.
- **.material** defines how a **GameObject** will draw itself. *Comment the line out and see what happens!*
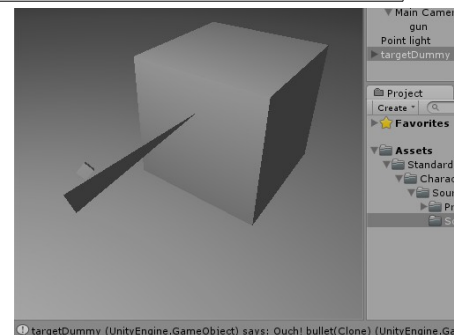
9. Wander Behavior

1 Create a new **C# Script**, name it "Wander", and use this code:

```csharp
using UnityEngine;
using System.Collections;

[RequireComponent (typeof(CharacterController), typeof(CharacterMotor))]
public class Wander : MonoBehaviour
{
    public float wanderTimer = 1, wanderSpeed = 5, rotateSpeed = 90;
    Vector3 wanderDirection;
    float timer;
    void FixedUpdate()
    {
        if (timer > 0) {
            timer -= Time.deltaTime;
        } else {
            Vector2 randomDir = Random.insideUnitCircle;
            wanderDirection = new Vector3(randomDir.x, 0, randomDir.y);
            wanderDirection.Normalize();
            timer += wanderTimer;
        }
        transform.rotation = Quaternion.RotateTowards(
            transform.rotation, Quaternion.LookRotation(wanderDirection),
            rotateSpeed * Time.deltaTime);
        CharacterMotor cm = GetComponent<CharacterMotor>();
        cm.movement.velocity = transform.forward * wanderSpeed;
    }
}
```

2 Add **Wander** to **targetDummy**, so it can move around by itself. Consider also adding **RespawnAfterFall**. If a **Character Controller** and **Character Motor** component did not automatically add to **targetDummy**, you need to fix errors in **Wander**, remove it, then re-add it, OR [Add Component] **Character → Character Controller** and **Physics → Character Motor** manually. ▶ Play-test to see **targetDummy** move!

- **CharacterController** is a component like **RigidBody**, except it is specifically for moving characters that travel up and down slopes. **CharacterMotor** is a component, already being used by the **First Person Controller**, that makes good use of the **CharacterController** component.
- Because **CharacterMotor** is not written using **C#**, **MonoDevelop** (and other editors) with auto-complete features don't help when using it! To know how to use the **CharacterMotor** script, you must read it yourself!
- **FixedUpdate** is like **Update**, except it is called when-the-physics-system-updates, not once-per-frame. **Update** is general, especially used for calculations effecting display. **FixedUpdate** is best for calculations changing *state*, like physics or AI. (**LateUpdate** is for *post-processing*, like camera-follow calculations)
- **Time.deltaTime** is the number of seconds passed since the last update. When using **Update**, **deltaTime** can be used to calculate the *frame rate* (how fast the game draws). For **FixedUpdate**, **deltaTime** is a constant value, which is how long each calculated game '*frame*' takes. These are different because *frame-rate* is often as-fast-as-possible, and a *frame* is ideally-always-the-same-duration (that's why it's called *Fixed*Update).
- **timer** uses a common pattern in game development, get comfortable with it!
- **randomDir** picks a random point from a *unit circle* around **targetDummy**, to move toward, which is converted to a 3D direction.
- A **Quaternion** is a rotation in 3D space. A **Vector3** can identify a direction being faced, but a **Quaternion** can identify a direction *faced* **and** which direction is *up*. **Quaternion** math and logic is more difficult than regular **Vector** math, but it is thankfully mostly done for you by the **Quaternion** class.
- **transform.forward** is a simple **Vector3** *unit vector* (has magnitude of 1) that gives the direction a **Transform** is facing. **.forward**, **.up**, and **.right** point like the hand pictured above points at **Z**, **Y**, and **X**.
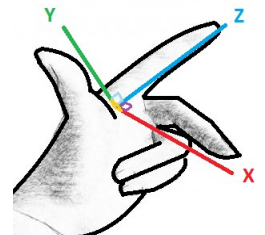
10. The Bullet-Through-Paper Problem

1 Change the **InitialVelocity** script (which is already a component of the **bullet**) to use this code:

```
using UnityEngine;
using System.Collections;

[RequireComponent (typeof (Rigidbody))]
public class InitialVelocity : MonoBehaviour {
    public Vector3 initialVelocity;
    void Start () {
        if (rigidbody.velocity == Vector3.zero)    rigidbody.velocity = initialVelocity;
    }
    void FixedUpdate() {
        float speed = rigidbody.velocity.magnitude;
        float distanceBetweenUpdates = speed * Time.deltaTime;
        if (distanceBetweenUpdates > 0) {
            Vector3 direction = rigidbody.velocity / speed;
            RaycastHit hitInfo;
            bool willCollide = Physics.Raycast(transform.position, direction,
                                     out hitInfo, distanceBetweenUpdates);
            if (willCollide && hitInfo.distance < distanceBetweenUpdates) {
                string label = name + "_hit_" + hitInfo.transform.gameObject.name;
                DrawFloatingLine(label, Color.red, transform.position,
                        transform.position + direction * distanceBetweenUpdates);
            }
        }
    }
    public static LineRenderer DrawFloatingLine(string label, Color color, Vector3 a, Vector3 b) {
        LineRenderer lr;
        lr = new GameObject(label).AddComponent<LineRenderer>();
        lr.SetVertexCount(2);        lr.SetWidth(0.01f, 0.01f);
        lr.SetPosition(0, a);        lr.SetPosition(1, b);
        lr.material = new Material(Shader.Find("Self-Illumin/Diffuse")); // won't export well!
        lr.material.color = color;
        return lr;
    }
}
```

• Note: **static** functions like **InitialVelocity.DrawFloatingLine** can be called by any other code. This particular function may be very useful when debugging 3D math problems in the future!

2 [Create ▾] a **Cube** named "wall" to shoot at, and make it thin (example: **Scale** 0.1, 1, 1).

3 Set the **Bullet Initial Speed** of the **Main Camera**'s **gun** to 100. [▶] Play-test to notice that when the bullets are fired at the **wall**, they usually pass through, but leave behind a red line. The length of the red line is the length traveled in one **FixedUpdate**.

• **Physics.Raycast** draws an imaginary line in the game engine, and stores the first thing hit by that line in the **RaycastHit** output. **FixedUpdate** uses that to consider the imaginary line between where a bullet is now, and where it will be next *frame*. If there is something in that *ray*, **DrawFloatingLines** makes a **red** line appear.

• The *Bullet-Through-Paper* problem is when a game engine does not trigger collision for a fast-moving object. This code doesn't solve the problem, it shows where the problem is probably happening (not taking 3D volume into account) with code you can re-write, which might be enough for most games. Truly solving the problem is hard, because of simultaneous motion and rotation of objects with mesh collision.
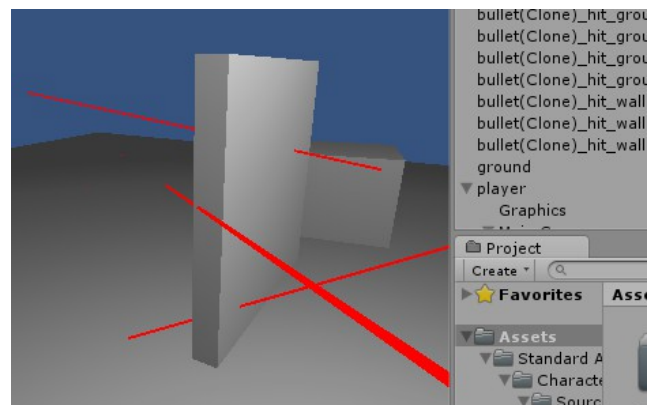
11. More!

- Unity3D has more tutorials on the official Unity3D website! Many go over the same basics covered by this and previous tutorials, and many more have new and interesting tutorial content! Take a look at these links:
    - Beginner lessons: http://unity3d.com/learn/tutorials/modules/beginner/scripting
    - Project Tutorials: http://unity3d.com/learn/tutorials/projects