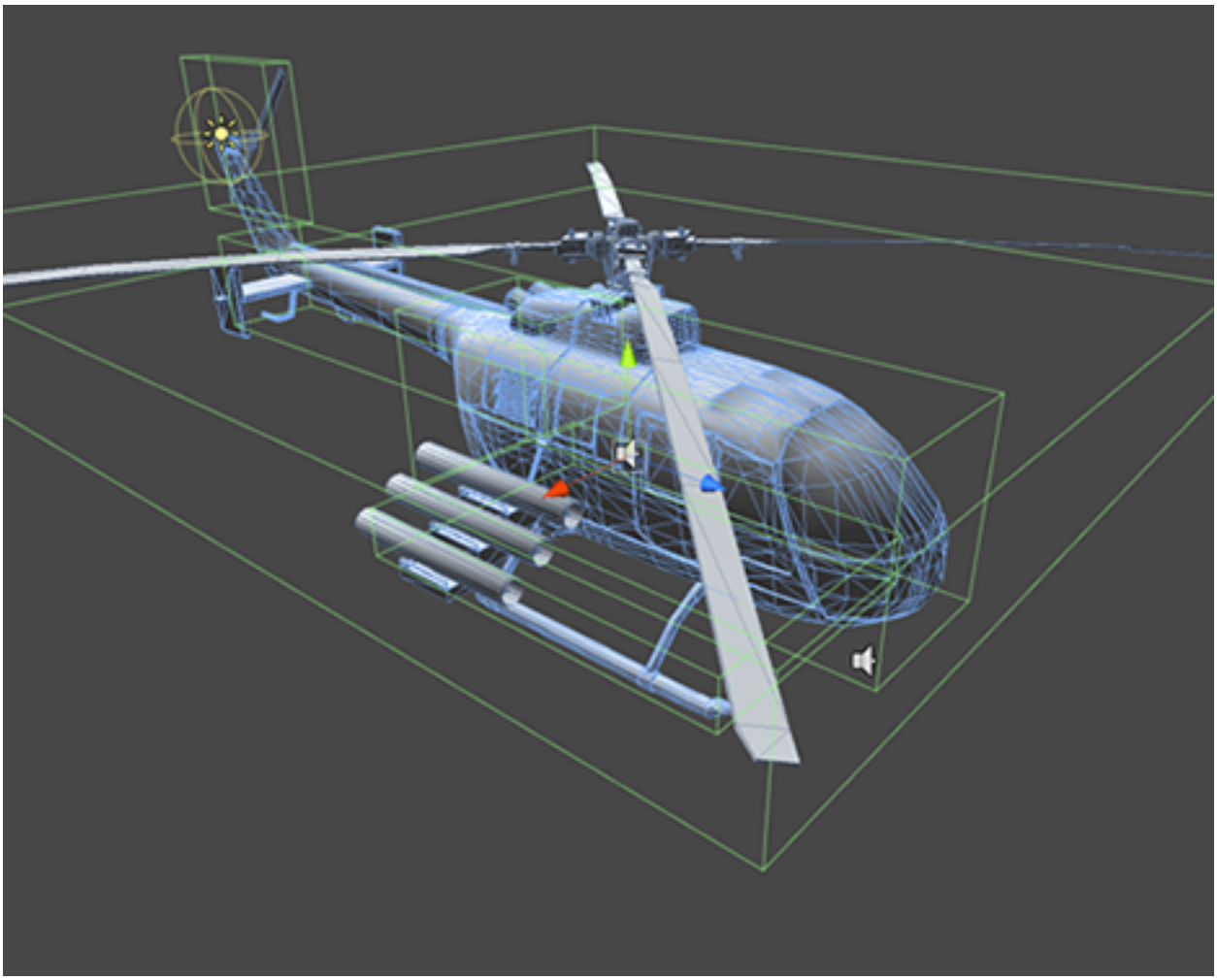
Unity3D

Helicopter Tutorial

A simple and easy to understand tutorial for the setup and programming of a helicopter in Unity3D

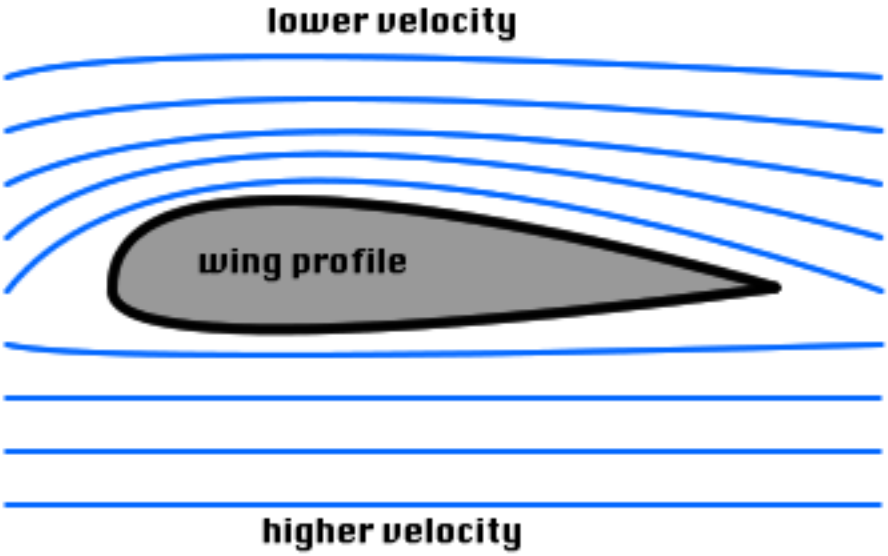


By Andrew Gotow

Table of Contents :

Part 1) How Helicopters Work

Helicopters are able to fly using the same basic principles as airplanes. A plane wing is what's called an *airfoil*, a long flat surface with a teardrop shaped profile. Essentially how an airfoil works, is by forcing the air on one side of the wing to flow faster than on the other.

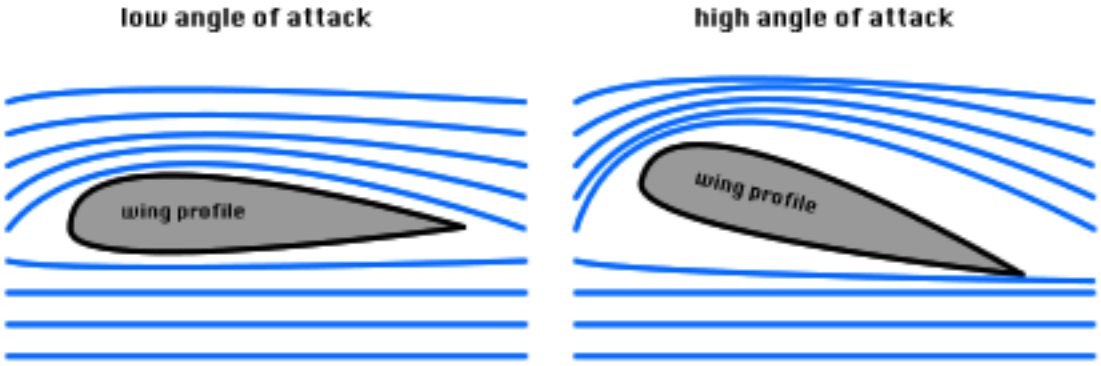


by changing the difference in wind speed, there is a pressure difference between the top and the bottom of the airfoil which exerts a force towards the area of lower pressure. Just like if you were to place an uneven object in a stream, the water would flow around one side slower than the other, and you would feel a pull towards that side.

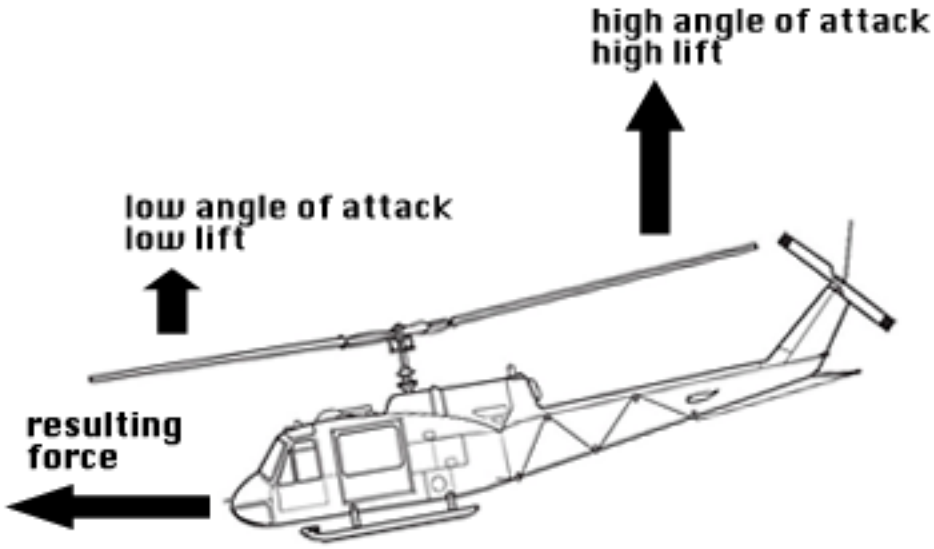
In order for an airfoil to generate lift, it needs to move very quickly, or have a very fast air current running past it in order to generate a high enough pressure difference to suck the vehicle upward. Airplanes achieve this by moving the entire vehicle very quickly forward so that the wind speed along the wing is high enough to lift the plane. Helicopters accomplish this in a very different way. Instead of moving the vehicle, helicopters move the wing! Each blade on the rotor is actually an airfoil, and they are spun rapidly around an axle so that the air speed between them is fast enough to lift the body of the aircraft off of the ground. By changing the speed of these rotors, it is also possible to alter the amount of lift the helicopter creates.

Now, how to turn this new vehicle… Here's another similarity between helicopters and airplanes. In order to turn, an airplane can lift small flaps on its wings, changing the angle of one of the wings. Because of this, the difference in

air speed is even greater than if the wing were directly facing the wind, and there is more lift on one side of the plane than on the other, causing it to roll away from that side. The angle the airfoil is at compared to the air flow is called *angle of attack*.



Helicopters use this effect to their advantage as well. Each rotor on a helicopter can tilt independently, allowing the pilot to change the angle of attack of each rotor blade as they spin. By increasing the angle of attack of the blades while they're on one side of the helicopter, you get a much higher lift on that side, forcing the helicopter to tilt and fly in the opposite direction.



By tilting the rotors in different ways, helicopters are able to fly forward, backward, left, and right, while facing the same direction.

Helicopters also have a *tail rotor*, a smaller propeller on the back end of the body, generally powered by a geared drive-shaft connected to the main engine.

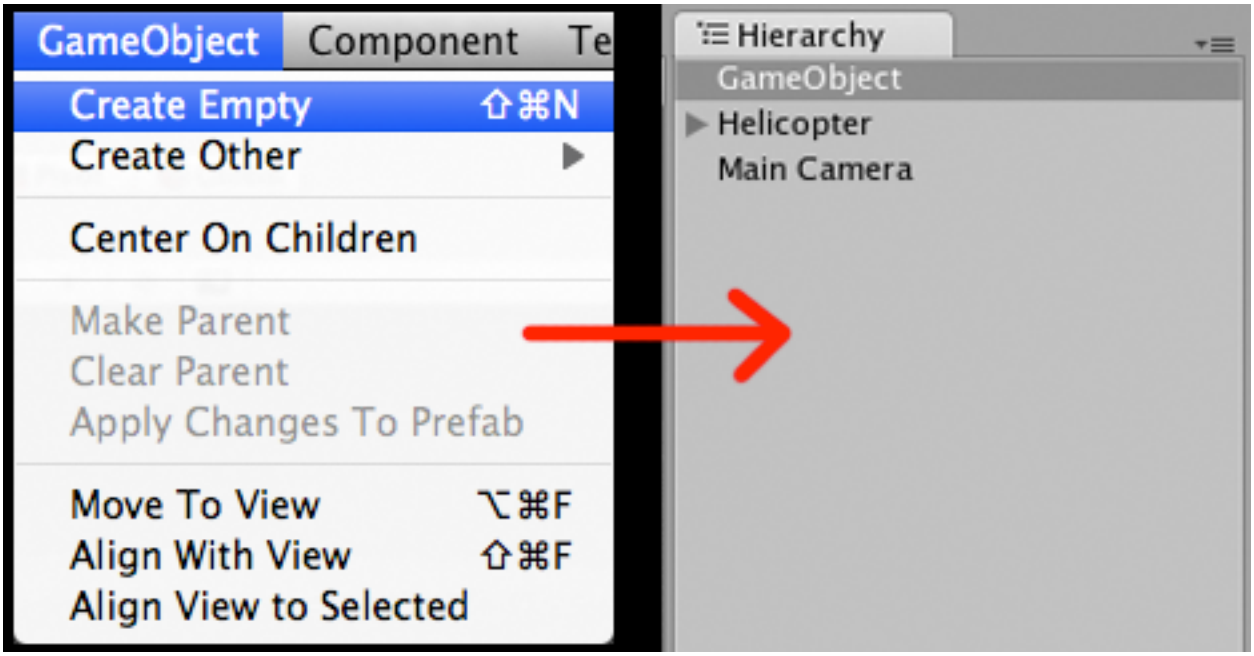
these rotors counteract all of the undesirable torque created when spinning the main rotor by spinning in the opposite direction. By adjusting the speed of the tail rotor, you can also increase or decrease the net torque on the helicopter, causing it to *yaw*, or spin on the Y axis. This allows the pilot to easily rotate the helicopter to face the desired direction while spinning in place, without having to bank, or make a wide turn like an airplane.

The combination of all of these systems allows a helicopter a full 6 degrees of freedom. It can fly forward, backward, left, right, up, down, and rotate along the X, Y, and Z axes, allowing much more control and maneuverability than any other flying machine in existence. However, if one of these systems fails, the results can be catastrophic.

Part 2) Setting up a Helicopter in Unity3D

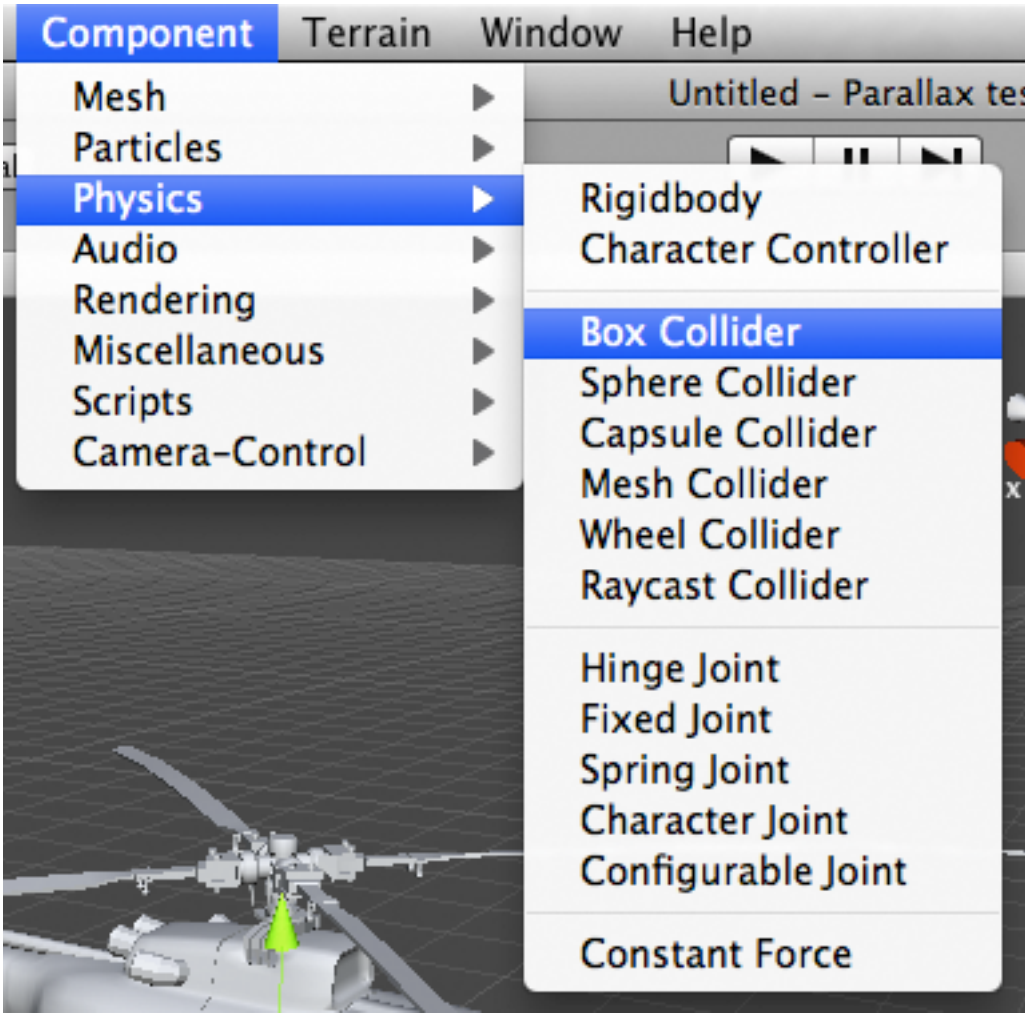
Now that we have a basic understanding of how real helicopters work, we have to figure out how to simulate it in Unity3d. While we could technically program all of the dynamics involved into the Unity engine, it would not run nearly fast enough, or be stable enough to control in a game scenario. In this tutorial, we will be constructing a simple rigidbody helicopter that manually applies the forces that the rotors would in real life by means of a simple script. Before we can do that however, we need to get the rigidbody set up properly.

First, we will need to create collision boundaries for our helicopter, so that the Physics engine knows the general shape of it. I usually do this by creating an empty game object to be used as a container, and then filling it with "Colliders". This is a fairly simple way to create a nice unevenly shaped collision boundary with multiple colliders while still keeping it organized.



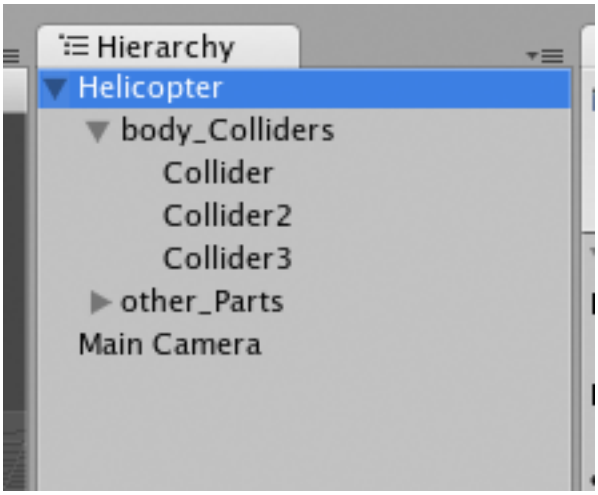
use the menu item *GameObject/CreateEmpty* to create an empty game object.

Now that you have a game object, rename it to something such as "colliders" or "collider container" so that you know what it is later on, and position it so that it is at (0,0,0) or wherever your helicopter object is in the world. You do not want to align all of your colliders only to find that they are actually at some arbitrary position when you parent them to your helicopter. Now create several other empty game objects, and attach Box, Sphere, or Capsule colliders to them.

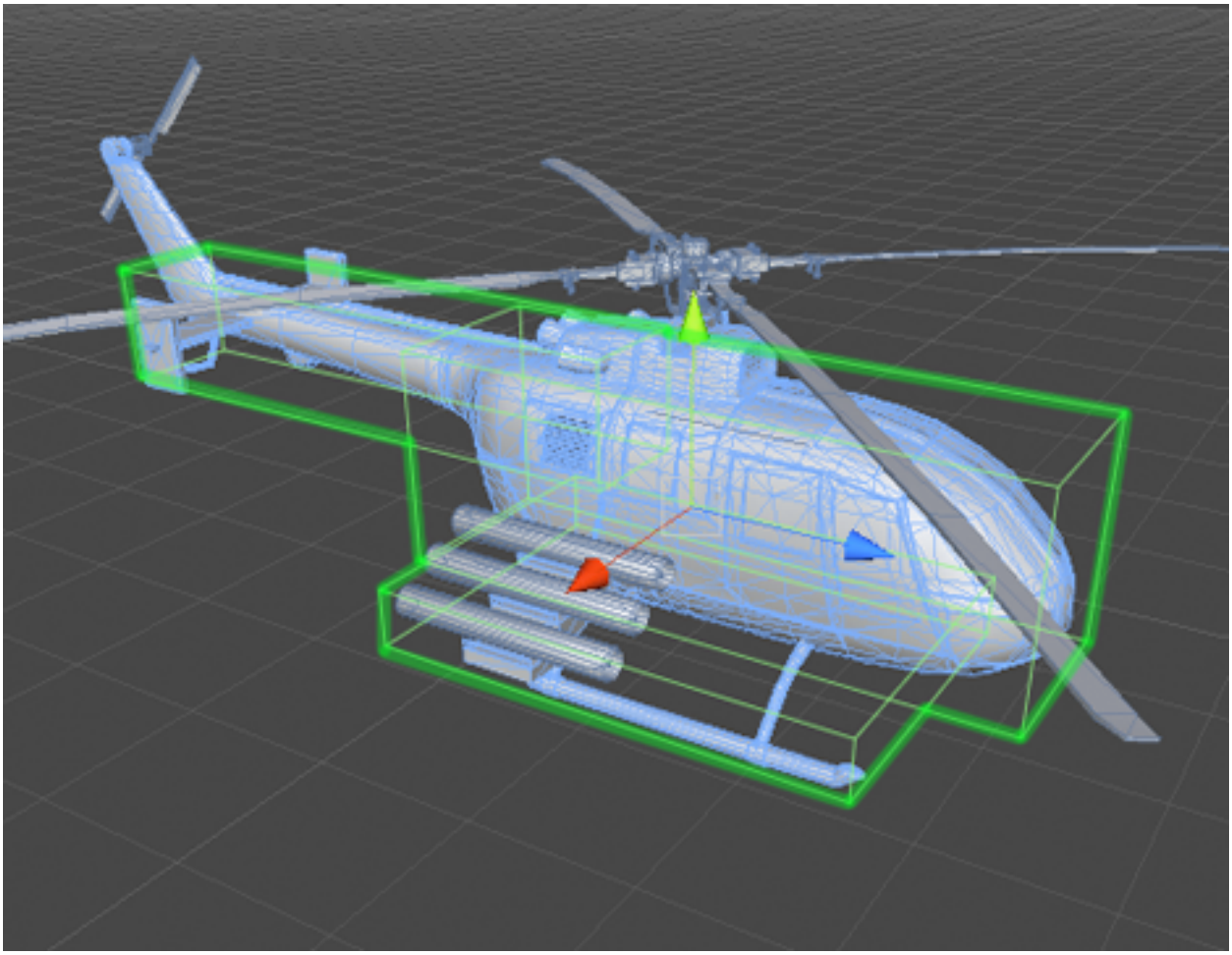


use the menu item *Component/Physics/ … Collider* to attach a collider

These are the actual collision boundaries in Unity, and determine how an object will collide with others. You can look at the colliders in the "Inspector", and edit their properties. You will want to change the sizes, and positions of the colliders (in the box sphere or capsule collider tag in the inspector), so that they match your helicopter's profile relatively well. They do not need to exactly match the helicopter, but just make a rough estimate that the player will not be able to notice. Once the entire set of colliders is set up, go ahead drag them inside the collider container that you set up earlier, (This will parent the collider objects to that object and will make them move as a whole) and finally, drag the container into your helicopter object. You should now have a nice clean hierarchy that you can easily read.

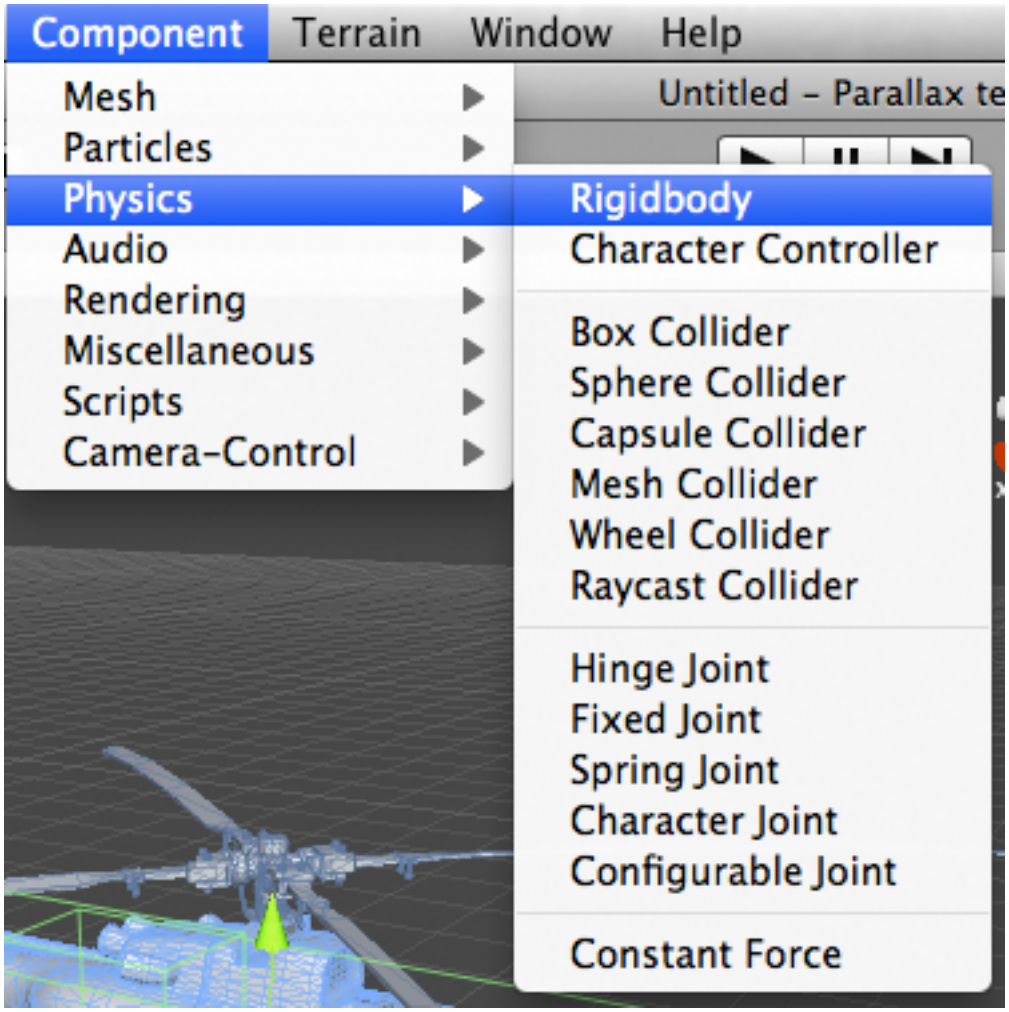


a sample hierarchy for colliders in a helicopter.



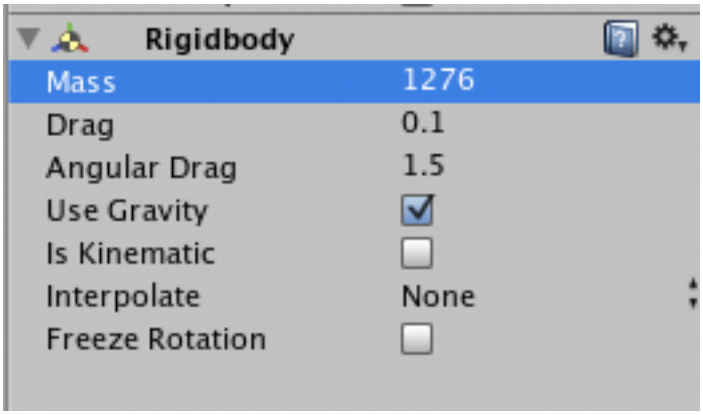
a picture of the final helicopter collision boundary, created with 3 box colliders.

Now that the collision boundaries are set up, we can actually attach the rigidbody component to our helicopter. Select your entire helicopter setup, and select the menu item *component/physics/rigidbody*. this will attach a simple rigidbody component, and will allow you to edit some of the more basic properties



the rigidbody menu item attaches a rigidbody to the selected object.

Now select your helicopter, and click the rigidbody tab in the inspector. Here you can change some of it's properties, such as mass, drag, and interpolation. We will set the helicopter's mass to a realistic value. All of the values in this tutorial were taken from an actual helicopter, the BO 105, which has a mass of around 1276 kilograms. simply type 1276 into the "Mass" text field, and it will set the rigidbody's mass to that value. Now we need to set drag values. While it is not necessary to do this, it makes the helicopter much easier to control because it decelerates the helicopter when the player is not applying a force to the body. I set the "Drag" property to around 0.1, and the "Angular Drag" property to around 1.5, but feel free to experiment with these values.

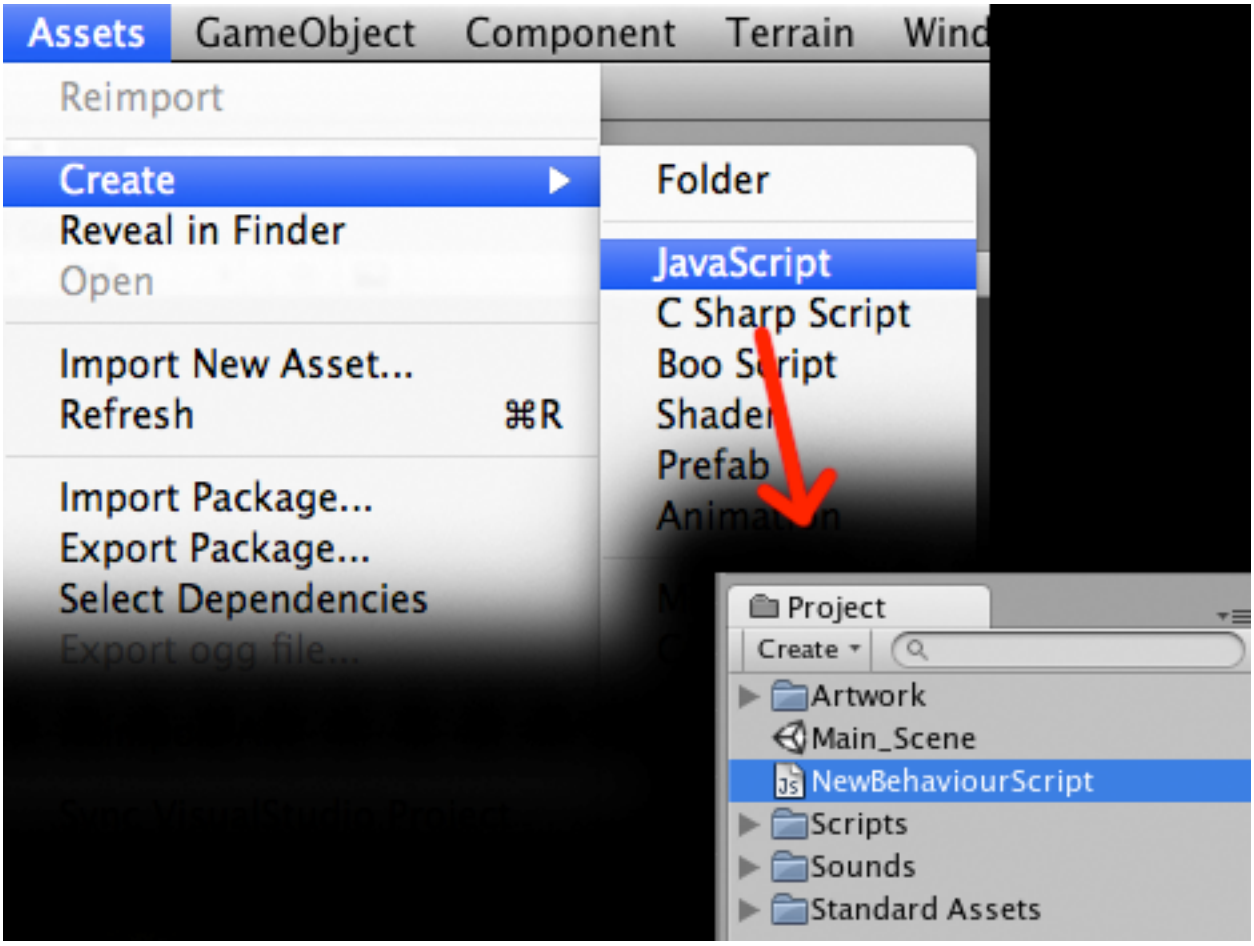


a snapshot of the properties for the helicopter rigidbody visible in the inspector.

And there you have it. When you press play, your helicopter should plummet to the ground and react accordingly. Next we will write a simple behavior script to allow it to actually fly using player input.

Part 3) Writing the Script

Ok, now that we have a helicopter set up in Unity, we will need to program it to respond to user input. First, we will need to create a new Javascript file. Go to the menu, and select *Assets/Create/Javascript*. This will add a new javascript file to your assets folder, that you can then edit and attach to game objects.



creating a new Javascript file and placing it in the assets folder.

Now double click on the script file and it will be opened int the default script editor. Here we can see the empty script file. This is where we will be writing all of the code for the helicopter controls. First off, we will need to define a few properties for our helicopter. We need to set up the maximum rotor force for both the main rotor, and the tail rotor, the rotor velocity, or throttle, which is used to determine the speed at which the rotor rotates and the amount of force they exert, as well as the sensitivity for the player controls, both forward, and sideways, so that we can use them later in the script.

A variable in Javascript is defined using the syntax

```
var max_Rotor_Force : float = 22241.1081;
      or
var name : type = default value;
```

so in order to define these variables, we can write the following at the top of our script.

```
var             main_Rotor_GameObject                : GameObject;
var             tail_Rotor_GameObject                : GameObject;

var             max_Rotor_Force                      : float = 22241.1081;
var             max_Rotor_Velocity                   : float = 7200;
static var      rotor_Velocity                       : float = 0.0;
private var     rotor_Rotation                       : float = 0.0;

var             max_tail_Rotor_Force                 : float = 15000.0;
var             max_Tail_Rotor_Velocity              : float = 2200.0;
private var     tail_Rotor_Velocity                  : float = 0.0;
private var     tail_Rotor_Rotation                  : float = 0.0;

var             forward_Rotor_Torque_Multiplier      : float = 0.5;
var             sideways_Rotor_Torque_Multiplier     : float = 0.5;

static var      main_Rotor_Active                    : boolean = true;
static var      tail_Rotor_Active                    : boolean = true;
```

The majority of these properties are simple enough to understand without significant explanation, but if you need clarification, look in the included player control script "Helicopter_Script" in the project folder.

Now that we have all the variables defined and set up properly, we can begin writing the basic script. First, we will write a simple FixedUpdate function to apply the rotor forces to the body of the helicopter. Fixed Update is a function that is automatically called by the Unity engine every physics timestep. This function is used to ensure consistency of forces applied or velocity alterations so that regardless of the frame rate the script will behave the same way. So at the end of the script, we can simply write

```
function FixedUpdate () {
}
```

and then add all the code we wish to use inside the brackets. First, we will start by calculating the torque applied to the body of the helicopter, so within the fixed update routine, we will write...

```
var torqueValue : Vector3;
```

TorqueValue is just an empty 3 dimensional vector representing the net torque on the object. We will fill this out later on in the script. Now we compute the "controlTorque", or the amount of torque applied to the body of the helicopter based on the player input. This is done to simulate the varying angle of attack on the helicopter blades, without having to do too much extra math.

```
var controlTorque : Vector3 = Vector3(
        Input.GetAxis( "Vertical" ) * forward_Rotor_Torque_Multiplier,
        1.0,
        -Input.GetAxis( "Horizontal2" ) * sideways_Rotor_Torque_Multiplier
        );
```

While this may seem confusing, it is really rather simple. The control input torque vector is equal to the input axes multiplied by the control sensitivity. The reason the Y value is set to 1.0, is because we want to simulate the torque on the body created by the spinning of the rotors. This is the easiest way to apply that force without adding too much extra code.

Now if the main rotor is active, then we wish to apply the net torque to the helicopter body as well as the lift force created by the spinning rotors, so we simply write...

```
if ( main_Rotor_Active == true ) {
        torqueValue += (controlTorque * max_Rotor_Force * rotor_Velocity);
        rigidbody.AddRelativeForce( Vector3.up * max_Rotor_Force * rotor_Velocity );
}
```

This applies the control torque to the net torque value, and also applies a force to the rigidbody in it's local Y direction based on the maximum speed, and the input throttle. In this statement, you can also add a stabilizing force to the body to make it slowly level out. This is important to keep the helicopter level and easy to control. The easiest way to do this is by using the *Quaterinion.Slerp()* function. This function simply spherically interpolates the rotation of the helicopter, from its current value to its rotation without the X and Z components rotations, essentially leveling it out while still keeping the original heading.

```
if ( Vector3.Angle( Vector3.up, transform.up ) < 80 ) {
        transform.rotation = Quaternion.Slerp( transform.rotation,
                        Quaternion.Euler( 0, transform.rotation.eulerAngles.y, 0 ),
                        Time.deltaTime * rotor_Velocity * 2 );
}
```

Finally, we need to apply the tail rotor's force to the net torque, and apply the torque to the helicopter body. So we check if the tail rotor is active, and then subtract the rotor's maximum force multiplied by it's throttle from the net torque and apply it relative to the body of the helicopter.

```
if ( tail_Rotor_Active == true ) {
        torqueValue -= (Vector3.up * max_tail_Rotor_Force * tail_Rotor_Velocity);
}

rigidbody.AddRelativeTorque( torqueValue );
```

That should be the end of the Fixed Update function, and should apply all the forces to the body accordingly. Now all that's left to do is write the throttle controls and add sound effects.

Much like the FixedUpdate function, the standard Update function is called automatically, but in this case it is called every frame, instead of just on physics timesteps. Usually the Update function is better for implementing controls because it makes the game much more responsive, because the input is checked every frame, instead of every 4 or so. So we begin this function the same way. We write

```
function Update () {
}
```

and within this we will write all of the responses to control input the user gives. First, we can animate the rotors. All this does, is spin the rotor gameObjects based on an ever increasing value.

```
if ( main_Rotor_Active == true ) {
        main_Rotor_GameObject.transform.rotation = transform.rotation *
        Quaternion.Euler( 0,
                          rotor_Rotation,
                          0
                          );
        }
if ( tail_Rotor_Active == true ) {
        tail_Rotor_GameObject.transform.rotation = transform.rotation *
        Quaternion.Euler( tail_Rotor_Rotation,
                          0,
                          0
                          );
        }

rotor_Rotation += max_Rotor_Velocity * rotor_Velocity * Time.deltaTime;
tail_Rotor_Rotation += max_Tail_Rotor_Velocity * rotor_Velocity * Time.deltaTime;
```

The rotors should now rotate nicely along their respective axes based on their maximum speed and the throttle of the helicopter. In order to make this helicopter stable and easy to fly, it is very important that the rotors will slowly drift towards neutral,

so that when the player releases a key, it will drift back to hovering. We can easily calculate the minimum necessary speed for the main rotor and tail rotor to keep the helicopter up in the air.

In order to find the main rotor minimum velocity, we simply do the math to find how much force is necessary to counteract the force of gravity, and then divide by the maximum force of the rotors. This will result in a value from 0.0 to 1.0, that is the minimum throttle needed to keep the helicopter stationary.

```
var hover_Rotor_Velocity = (rigidbody.mass * Mathf.Abs( Physics.gravity.y )
                                / max_Rotor_Force);
```

The tail rotor velocity can also be easily solved for. Because the sole purpose of the tail rotor is to counteract the torque of the main rotor, just find the amount of torque created by the main rotor, and use that to determine the tail rotor throttle.

```
var hover_Tail_Rotor_Velocity = (max_Rotor_Force * rotor_Velocity)
                                    / max_tail_Rotor_Force;
```

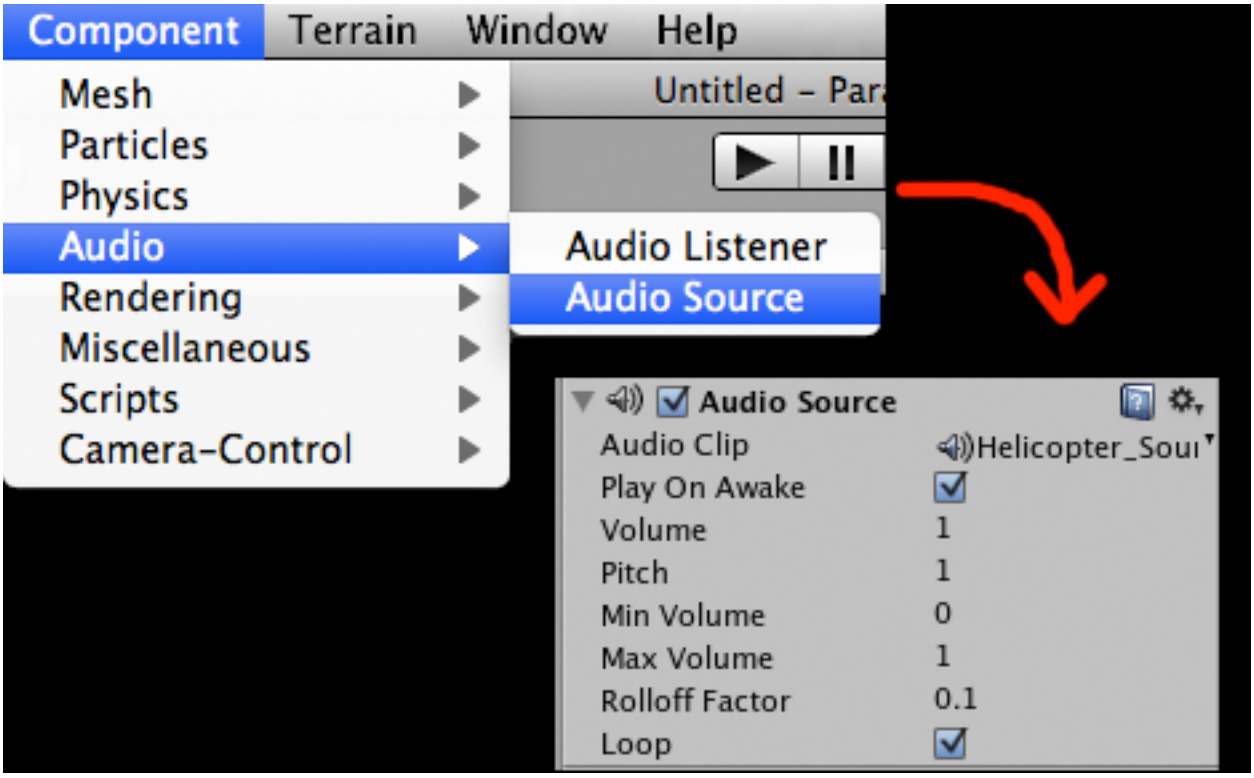
Now, if the player is pressing the key to increase the rotor throttle, then increase the throttle of the main rotor, otherwise, slowly interpolate it back to the hover velocity, so that it will hover in place. And lastly, set the tail rotor velocity to the minimum velocity, and then increase or decrease it by the player input. This will make the tail rotor spin slower or faster when the player presses "Left" or "Right", making it apply an unbalanced torque and spinning the helicopter in that direction.

Lastly we limit the rotor velocity to a value between 0.0 and 1.0, just to make sure we can not possibly apply a force greater than either.

```
if ( Input.GetAxis( "Vertical2" ) != 0.0 ) {
        rotor_Velocity += Input.GetAxis( "Vertical2" ) * 0.001;
}else{
        rotor_Velocity = Mathf.Lerp( rotor_Velocity,
                                    hover_Rotor_Velocity,
                                    Time.deltaTime * Time.deltaTime * 5
                                    );
}
tail_Rotor_Velocity = hover_Tail_Rotor_Velocity - Input.GetAxis( "Horizontal" );

if ( rotor_Velocity > 1.0 ) {
        rotor_Velocity = 1.0;
}else if ( rotor_Velocity < 0.0 ) {
        rotor_Velocity = 0.0;
}
```

Lastly, the helicopter should make a sound. This is relatively simple, because we already have a value from 0.0 to 1.0 that corresponds to the throttle, all we need to do is attach an audio source, and adjust its pitch. In unity, you Attach an audio source just like you attach any other component. Go to the component menu, and select *Component/Audio/Audio Source.* This will simply attach a 3D audio source to the object you have selected, in this case our helicopter.



Like all components, the audio source will show up in the inspector view, and you can edit it's properties. Set the audio clip to whatever sound you wish to make your helicopter play, then set play on awake, and loop to true. This will make the audio source play the helicopter engine sound constantly throughout the entire game.

Now back in your control script, simply add a new line in the Update function that says

```
audio.pitch = rotor_Velocity;
```

and your audio should increase and decrease in pitch depending on the speed at which the rotors are spinning.

Don't forget to attach the script to the helicopter object by simply dragging it from the assets folder, onto the helicopter body.

If you would like, you can make other scripts for different parts of the helicopter, such as a simple script that deactivates the rotors if they come in contact with

something, such as those included in the Unity project folder. These are entirely optional, and so I will not go into depth in this tutorial, but they are very useful if you are making a helicopter combat game, or something similar.

Part 3) Making a HUD

Now if you were to fly our helicopter right now, you would find that it is relatively difficult to land, because you have no indication of your altitude, or throttle. What we need is a set of gauges. By far, the simplest way to do this is by using the built in Unity GUI in a script. attached to the helicopter, or to another game object. In this example, we will have a game object specifically dedicated to drawing the HUD.

Create an empty game object as before, and attach a new javascript. This will serve as the main script for our HUD. The built in Unity3d GUI is very easy to use. You simply supply a texture or a string, and a rectangle to draw it in, and using the GUI.Label function, you can draw it anywhere onscreen. This also works for buttons, sliders, edit fields, and anything else you can think of, but for this application we only need to use labels. Now open the new script and begin editing. First, we need to specify what the player object is so we can read it's throttle values, and draw onscreen textures to represent them. This is accomplished by defining some variables in the script. By simply writing the following code at the top of the file, we can set the script to draw everything we need.

```
var             player_gameobject       : GameObject;
var             altimeter_texture       : Texture;
var             throttle_texture        : Texture[];

private var     helicopter_throttle     : float;
```

These variables specify the textures to be used as the gauges and the game object to read the values from.

Next we need to actually draw these gauges. Unity3D GUI calls must be made in the OnGUI function, so just write the following in place of the default update function.

```
function OnGUI () {
}
```

Now it's easy to write the necessary code! In the OnGUI function we must perform a simple raycast to determine the altitude of the helicopter. Create a Raycast Hit property, and then use the Physics.Raycast function to find the distance to the ground.

```
var groundHit : RaycastHit;

Physics.Raycast( player_gameobject.transform.position - Vector3.up,
                -Vector3.up,
                 groundHit
                 );
```

This raycast will fill out the groundHit variable with all of the collision information, so we can determine the distance to the nearest thing under you. Next we set the helicopter throttle value to the rotor_velocity variable of the helicopter control script.

```
helicopter_throttle = player_gameobject.GetComponent
( "Helicopter_Script" ).rotor_Velocity;
```

This simply finds the script attached to the player gameobject called "Helicopter_Script", and then reads the value of the variable rotor_Velocity.

Now all that's left to do is display them on the screen. First we draw the backdrop for the altimeter, and the throttle gauge, then we draw labels overtop. You may have noticed that when we defined the texture for use with the throttle gauge, we set up an array. Well, this is because the built in Unity3D GUI does not support sprite rotation, so we must make multiple frames to flip between to display the different values. In this example, there are simply 10, but that number can easily be changed.

```
GUI.Label( Rect( 0, 0, 128, 128 ), altimeter_texture );
GUI.Label( Rect( 0, 128, 128, 128 ), throttle_texture[ helicopter_throttle * 10 ] );
```

Because helicopter_throttle is a value between 0 and 1, we can set the frame of the throttle gauge animation to be the helicopter_throttle value * the number of frames (in this case 10).

And now simply draw some labels, and print the altimeter value on top of the gauge backdrop.

```
GUI.Label( Rect( 40, 40, 256, 256 ), Mathf.Round( groundHit.distance ) + " m" )
GUI.Label( Rect( 20, 182, 256, 256 ), "ENG" );
```

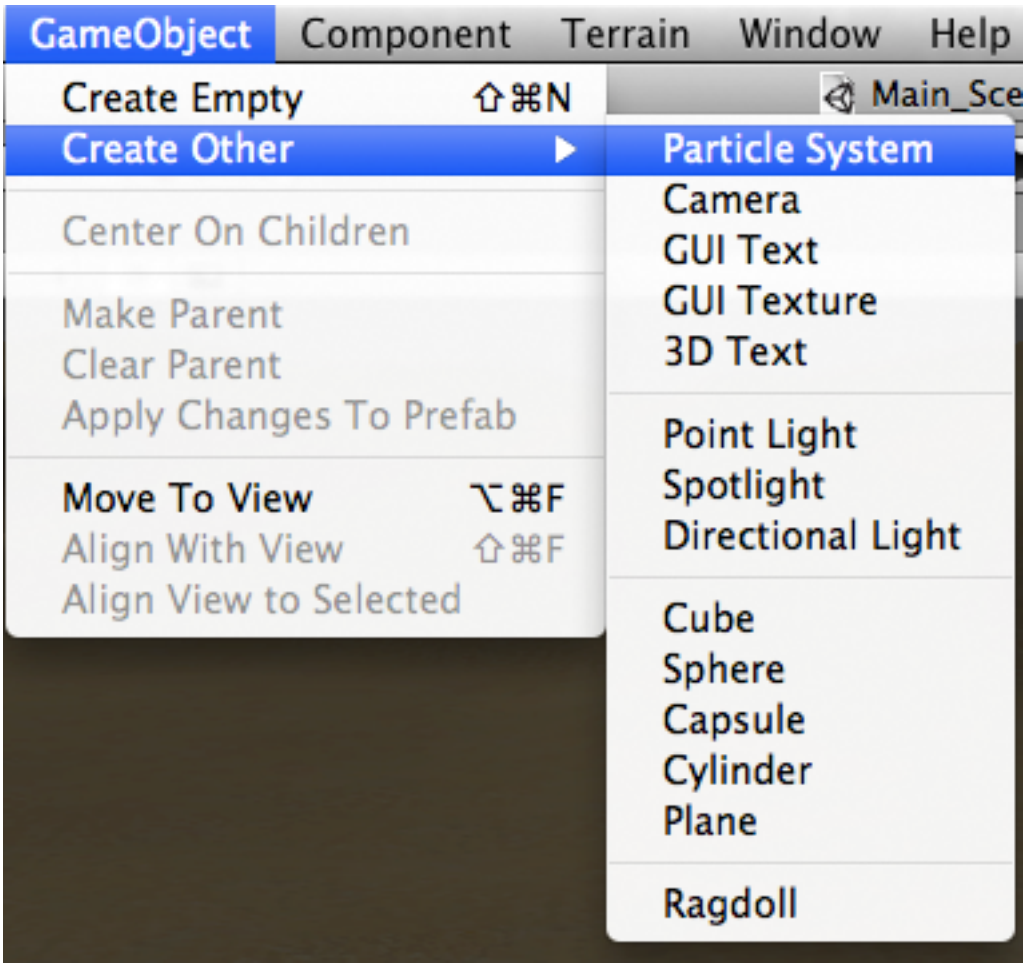
And that should be it! Just attach this script to any game object in the scene, and set the variables correctly, and you should have a simple, and functional HUD for your helicopter.

Part 5) Attaching Weapons

If you're making a helicopter, it almost undoubtedly contains guns, so I figured I'd write a simple weapon script in this tutorial, to show the basics of raycasts and the uses of the RaycastHit object. For this script, we're going to need a few variables, mainly one to tell the delay between shots, another to count up to that delay, another for the effect that is created when something is hit, and a fourth for the gun itself.

```
var             gun_Emitter_Object      : GameObject;
var             weapon_Fire_Delay       : float = 0.0;
var             bullet_Impact_Prefab    : GameObject;
private var     weapon_Fire_Timer       : float = 0.0;
```

Now, the gun emitter object should be a particle emitter, with an audio source attached. This will serve as the effect created when the gun fires, as well as the source of the gunshot sound. To create a basic particle emitter, select the menu item, Gameobject/Create Other/Particle System, and it will make a new game object that emits particles. Tweak it's values until you have something you like, and attach an audiosource component.



Now, in the update function of the new gun script, we can write some simple code. First, we will want to turn off the particle emitter, then check if the fire key is down and the time since the last gunshot is greater or equal to the fire delay. If so, set the fire timer to 0, tell the gun object's audio source to play, and set it to emit particles.

```
function Update () {
	gun_Emitter_Object.particleEmitter.emit = false;
	if ( Input.GetButton( "Fire1" ) && weapon_Fire_Timer >= weapon_Fire_Delay ) {

		weapon_Fire_Timer = 0.0;
		gun_Emitter_Object.audio.Play();
		gun_Emitter_Object.particleEmitter.emit = true;
	}
}
```

Also, within this if statement, write the raycast code. We set up a Raycast Hit object, and then call the Physics.Raycast function to fill in all of the hit information. If the raycast function returns true (if there's something in the line of fire), then we can create a bullet hit prefab, and do whatever else you may want to do when something gets shot.

```
var hit : RaycastHit;

if ( Physics.Raycast( gun_Emitter_Object.transform.position,
			gun_Emitter_Object.transform.forward,
			hit )
			) {

	Instantiate( bullet_Impact_Prefab,
			hit.point,
			Quaternion.LookRotation( hit.normal )
			);
}

weapon_Fire_Timer += Time.deltaTime;
```

And lastly, increment the fire delay timer, like we are in the line above. You should now be able to attach this script to any game object in the scene, specify a gun object and a bullet impact prefab asset, and then be able to fire.

Conclusion

Using the Unity3D Physics engine, it is easy to create a simple helicopter script based largely off of real life helicopters. All of these scripts can be modified to fit almost any application. Different values can be substituted in for the script variables, such as the maximum rotor force, and the tail rotor force to change the performance of the helicopter. The weapon script can also be easily modified so that it will behave with existing games or entirely new systems. The included project folder for this tutorial contains all of the contents of this tutorial, including well documented and commented scripts, as well as some simple, royalty free artwork.

If you experience any problems with this tutorial, don't hesitate to ask for help.

Feel free to contact me at **AndrewGotow@Gmail.com**.
You can download this tutorial free off of my website, **www.Gotow.net/Andrew/Blog**.