**3D Action Template** is a simple template for creating action (or quest) games based on C# components.

Includes:
- Menu based on 3D scene objects
- Concepts of "game" and "mission" (divided into sub-tasks)
- HUD
- First-person controller (running, jumping, crouching)
- Inventory, item and weapon pickup and usage
- Melee and ranged weapons
- Vehicles and airplanes controlled by the player
- AI enemies that can patrol and attack

**Installation**: Before importing the package, delete the **bake_lighting** and **csharp_template** folders of your C# project to avoid file collisions (otherwise the lighting will be broken and the character will not move as intended)!

To launch the demo game, select the "**RunDemo**" launcher in the editor, or open the world: "data/AlexanderPanichev/3DActionTemplate/sample/worlds/main_menu.world" and click "Play".

To launch the sample, select the "**RunSample**" launcher (or open the world ".../worlds/sample.world").

# Architecture

Interaction between components is implemented via events (**System.Action**). Some components **generate events**, others **listen** for them.

Examples:
1) “**Button**” and “**Exit Game**”. The first creates an event (triggers on click), the second listens and closes the game.
2) “**Player enters room**” and “**Spawn enemies**”. The first is a trigger that creates an event, the second listens and spawns enemies.
3) “**Door handle**” and “**Open door**”. The handle creates an event when turned. The door opens if it has a component listening for that event. With this approach, doors can open in any way you configure: entering a room, killing all enemies, pressing a button in another corner, etc.
4) “**Player health**” and “**Game Over**”. Health can generate different events: health depleted, fully restored, below 50%, etc. Game Over listens for the "health depleted" event.

In code: there's a **CEventHandler** class that all listening components inherit. The virtual method **Activate**() is used to handle incoming events:

CEventHandler.cs

data > AlexanderPanichev > 3DActionTemplate > template > components > common > CEventHandler.

```
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using Unigine;
5
6  [Component(PropertyGuid = "60f410ec0e9dd75be62424feefd008451c07dce0")]
7  public class CEventHandler : Component
8  {
9      public Action onActivated;
10
11     public virtual void Activate(Component sender) {}
12
13     public void OnReceiveEvent(Component sender)
14     {
15         Activate(sender); // notify derived class
16         onActivated?.Invoke(); // notify subscribers
17     }
18 }
19
```

Components that generate events use public **List<CEventHandler>** to assign receivers directly in the Editor:

CPhysicalTrigger.cs

data > AlexanderPanichev > 3DActionTemplate > template > components > quest > CPhysicalTrig

```
4  using Unigine;
5
6  [Component(PropertyGuid = "957f398422b3771321d891566fbc98bdf85302f8")]
7  public class CPhysicalTrigger : Component
8  {
9      public enum Mode
10     {
11         Single,
12         Multiple,
13     }
14     public Mode activation_mode;
15     public List<CEventHandler> onEnter = new List<CEventHandler>();
16
```

When the event occurs, it notifies all assigned receivers.

```
// notify subscribers
foreach (var receiver in onEnter)
    receiver.OnReceiveEvent(this);
```

All components are located at:
**data/AlexanderPanichev/3DActionTemplate/template/components/**
Divided into 5 folders:
1) **common** – basic player components (player control, HUD, game)
2) **main_menu** – menu creation components (buttons)
3) **quest** – quest game components (doors, items, triggers)
4) **shooter** – shooter components (weapons and enemies)
5) **vehicles** – vehicle components (cars, planes, exoskeletons)

# Common components

**CEventHandler** – base class to inherit from and override Activate() to handle events. Used in the demo to track player progress (see «game» node and CGame.cs).

**CGame** – singleton storing references to player, health, and mission tasks.

**CHUD** – minimal HUD interface. Shows current objective, hints, health.

**CPlayer** - "the player" (or rather, their data). Links to camera, weapon/item mount points, current health, and inventory. Allows interaction with the world: pick up and use items. Should include CHealth somewhere in the hierarchy.

**FirstPersonController** – player movement: walking, running, jumping, crouching, camera control.

**UI** – wrapper for engine widgets, adds pivot, anchor, and auto-scaling based on resolution. **Strongly recommend using the Toolkit add-on from Unigine Add-on Store instead.**

# Main_menu components

**CChangeCamera** – switches the active camera when triggered.

**CRunConsoleCommand** – runs a console command on event (e.g., loads another world)

**CUIButton** – attached to **Unigine.Object** (e.g., ObjectMeshStatic). A button that fires an event on click.

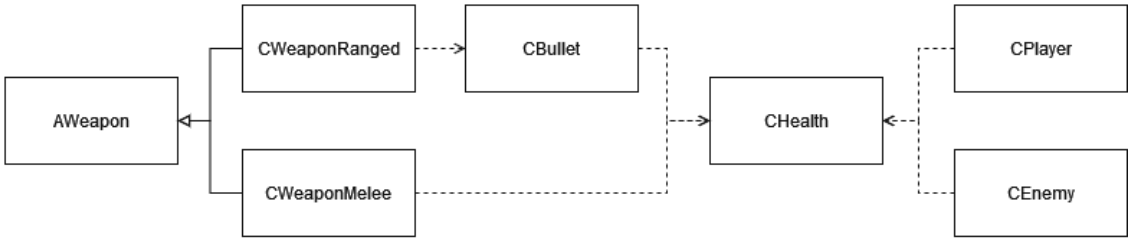
# Quest components

**CDoor** – a door that opens/closes on event.

**CInteractable** – interactive object that can be picked up, used, or requires a held item to use. Fires OnInteract. Must be of type **Unigine.ObjectMeshStatic**. **Item Mask** must match **CPlayer**'s for interaction.

**CPhysicalTrigger** – trigger that fires OnEnter when the player enters. Must be assigned to a **Unigine.PhysicalTrigger** object.

# Shooter components




**AWeapon** – abstract base class for weapons. CWeaponMelee and CWeaponRanged inherit from it.

**CBullet** – component attached to bullet/projectile objects.

**CDestroyTimer** – destroys object after "timer" seconds (used by bullets).

**CEnemy** – enemy with patrol, attack, and idle modes. Assign to **Unigine.Object** with a **Rigid Body**.

**CHealth** – handles unit health. Used by players and enemies. Weapons interact only with this component.

**CKillEnemiesTask** – fires OnKillAll when all enemies in the list are dead.

**CWeaponMelee** – melee weapon.

**CWeaponRanged** – ranged weapon.

# Vehicles components

**CAirplane** – airplane controller. Assign to **Unigine.Object** with **Rigid Body**.

**CCar** – car controller. Assign to **Unigine.Object** with **Rigid Body**.

**CKeyPressed** – event trigger for keypress.

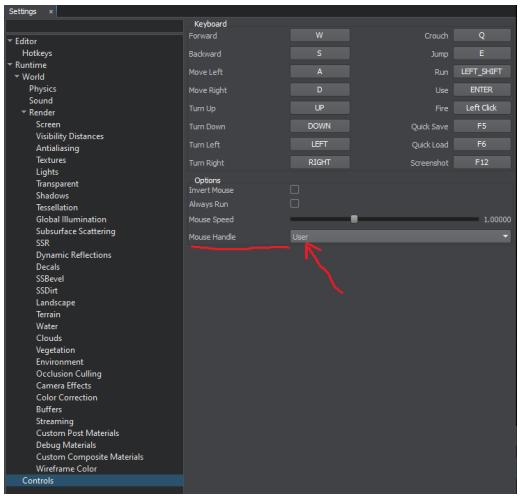
**CSwitchPlayer** – switches player types on event.

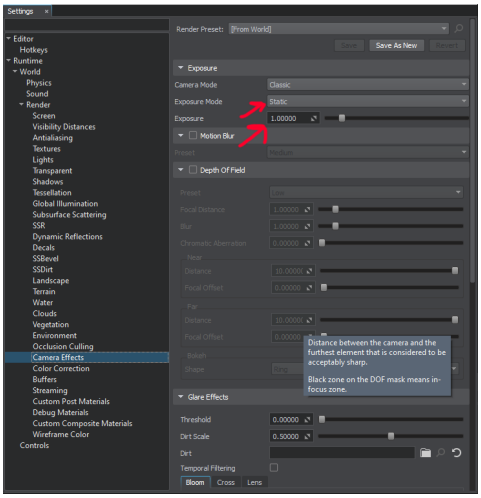
# How Main Menu Works

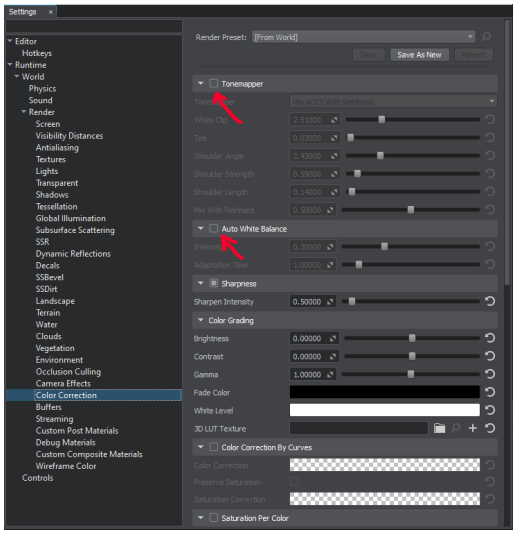
Strongly recommend using the **Toolkit** add-on from Unigine Asset Store: https://store.unigine.com/add-on/1f05cd6d-db59-6eee-a4f7-9ba129dedac0/description

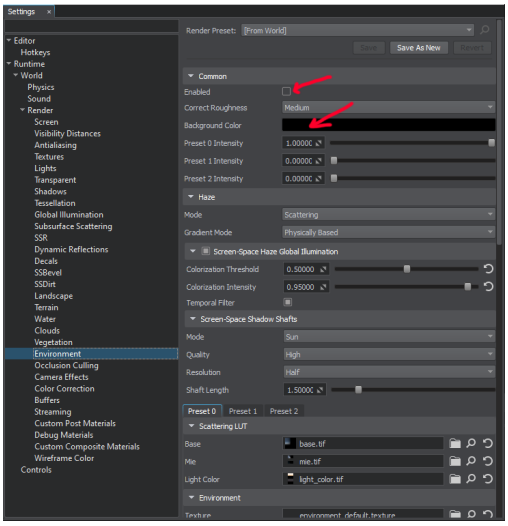
This example uses scene objects. Buttons are **ObjectMeshStatic** placed in front of the camera. Each frame, rays are cast from the camera through the cursor. If intersecting, the object is considered hovered.

First, configure the scene. Disable all environment and set the cursor to "always visible and not captured" (MouseHandle.User).

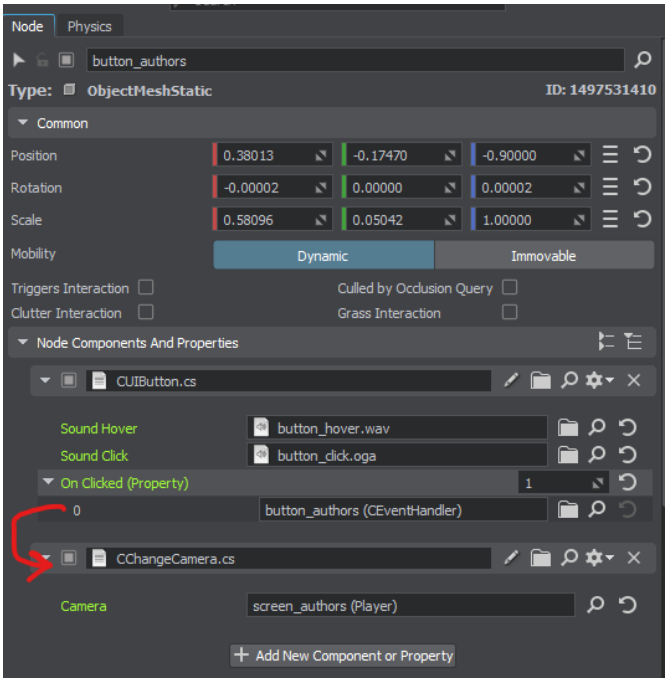


Now the button:

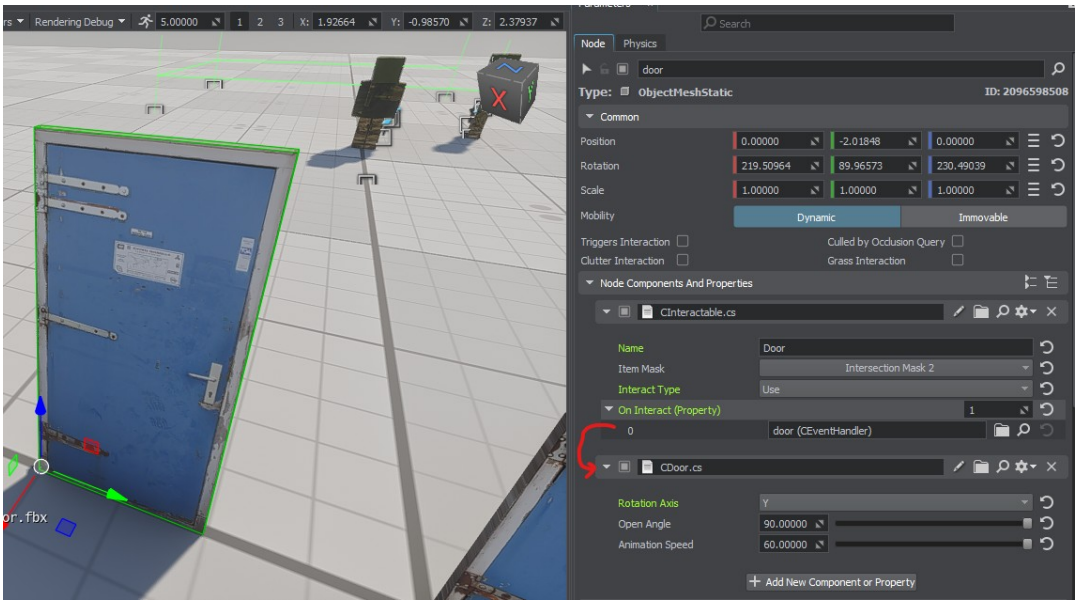



It's a **ObjectMeshStatic** (plane) facing the camera. It has **CUIButton** and **CChangeCamera**. When hovered and left-clicked, it fires OnClicked, received by CChangeCamera (on the same node), which switches the camera.

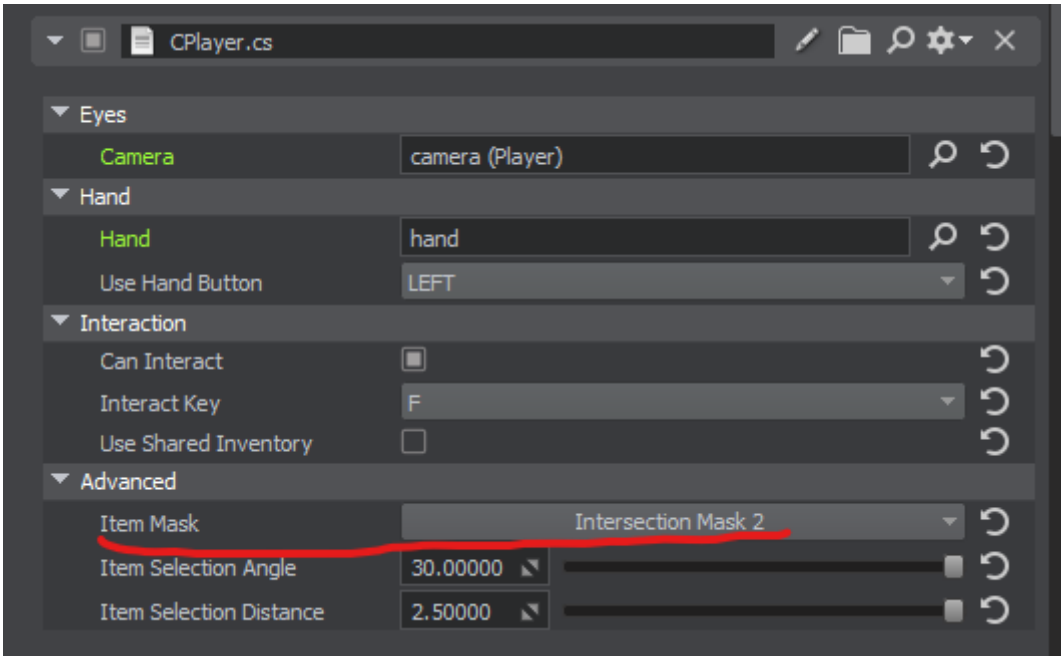
All other buttons work the same.

# How Doors Work

A door is an **interactive** object. Similar logic to menu buttons.




It uses two components: **CInteractable** and **CDoor**. The first marks it usable by the player. Make sure the **Item Mask** matches the one in **CPlayer:**

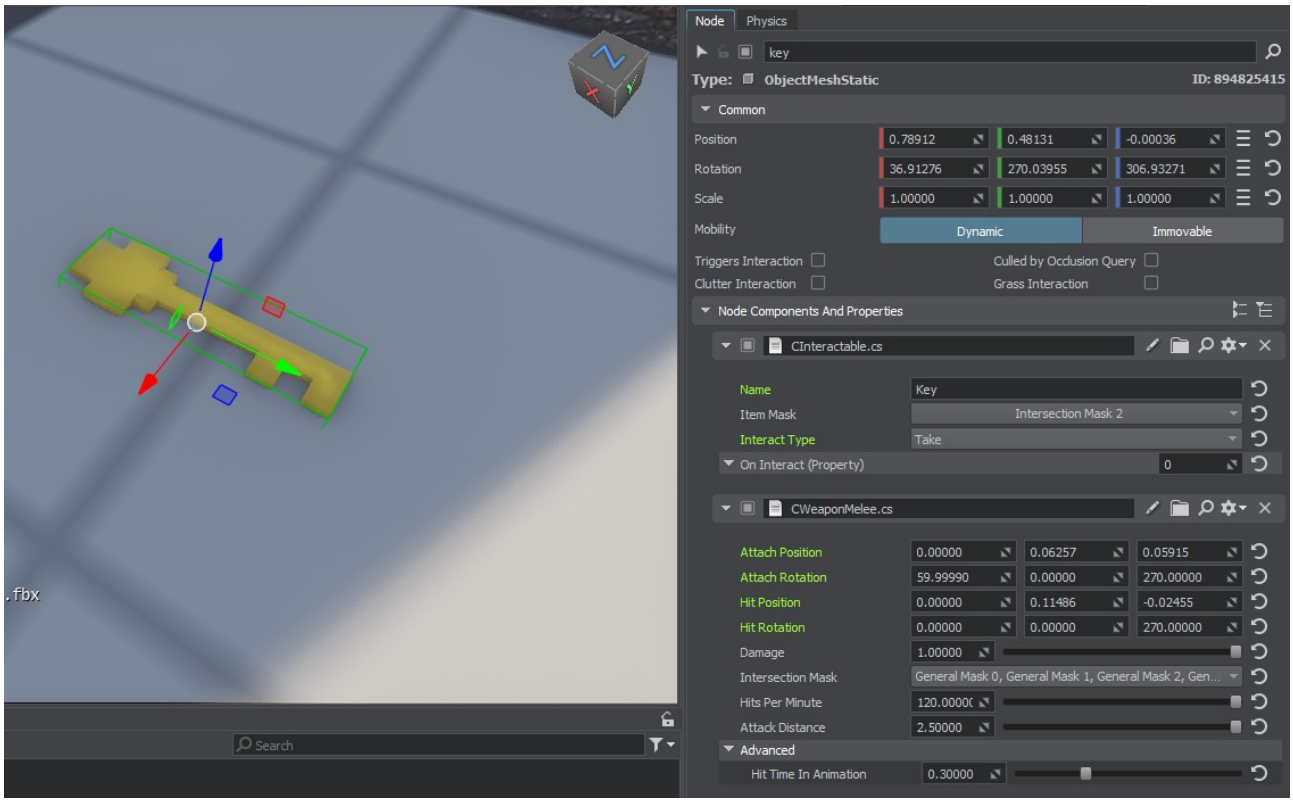



Also, enable "**Can Interact**" to allow interaction logic.
**Item Selection Angle** – viewing angle threshold for interaction.
**Item Selection Distance** – maximum distance for interaction.

Okay, let's back to door. Pressing F triggers OnInteract, which is handled by CDoor, performing the open/close animation.

# How Weapons Work

Take the key from the sample, which also serves as a melee weapon.

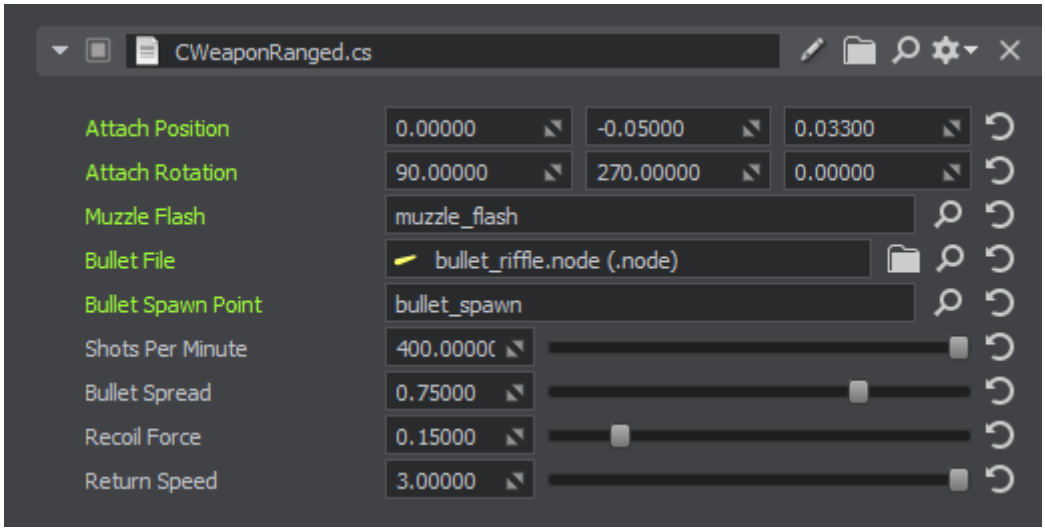



It has **CInteractable** and **CWeaponMelee**. The first allows picking it up (if **Interact Type** is “**Take**” and if the **Item Mask** matches **CPlayer’s**).

The second component describes the weapon:

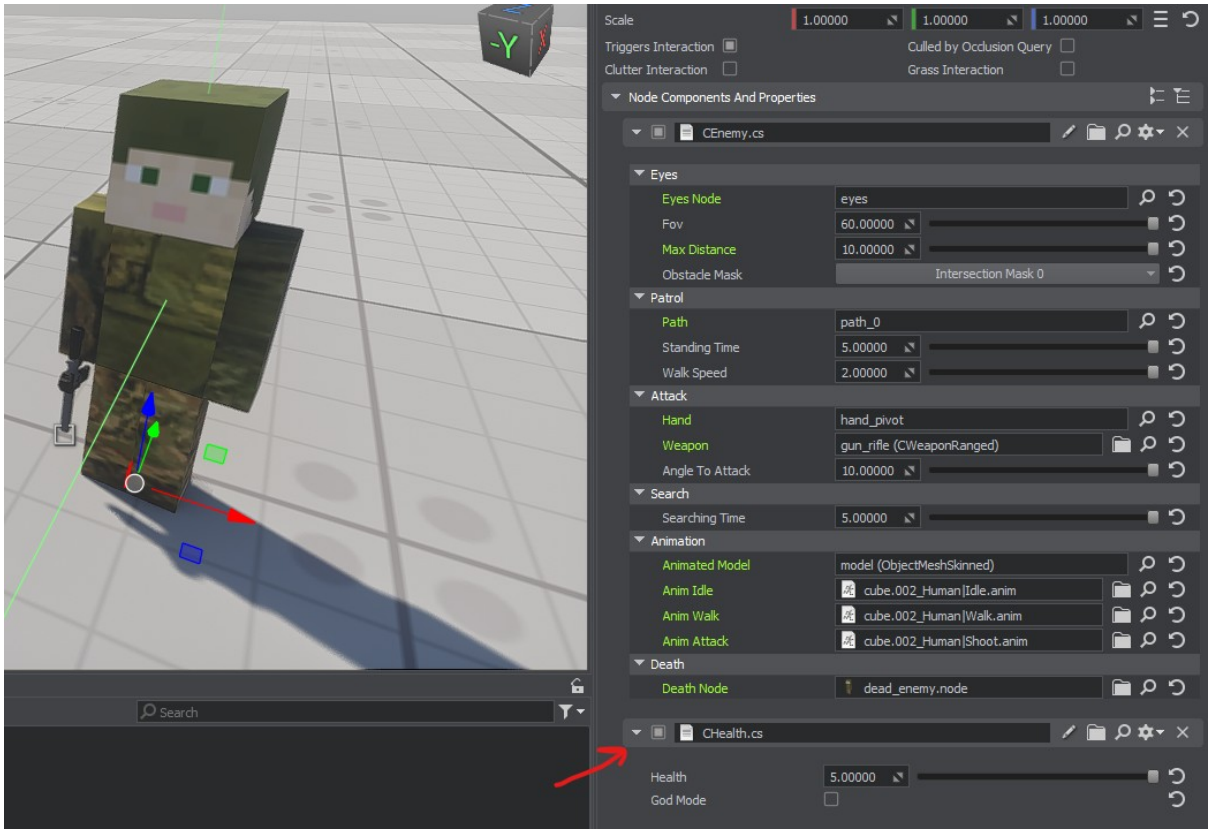
**Attach Position/Rotation** – where the weapon attaches to the player’s hand node.
**Hit Position/Rotation** – weapon pose at the middle of «attack animation».
**Damage** – damage value.
**Intersection Mask** – enemy types that take damage (must match surface masks)
**Hits Per Minute** – rate of attack
**Attack Distance** – max reach

Ranged weapons (guns, bows, etc.) use **CWeaponRanged**:



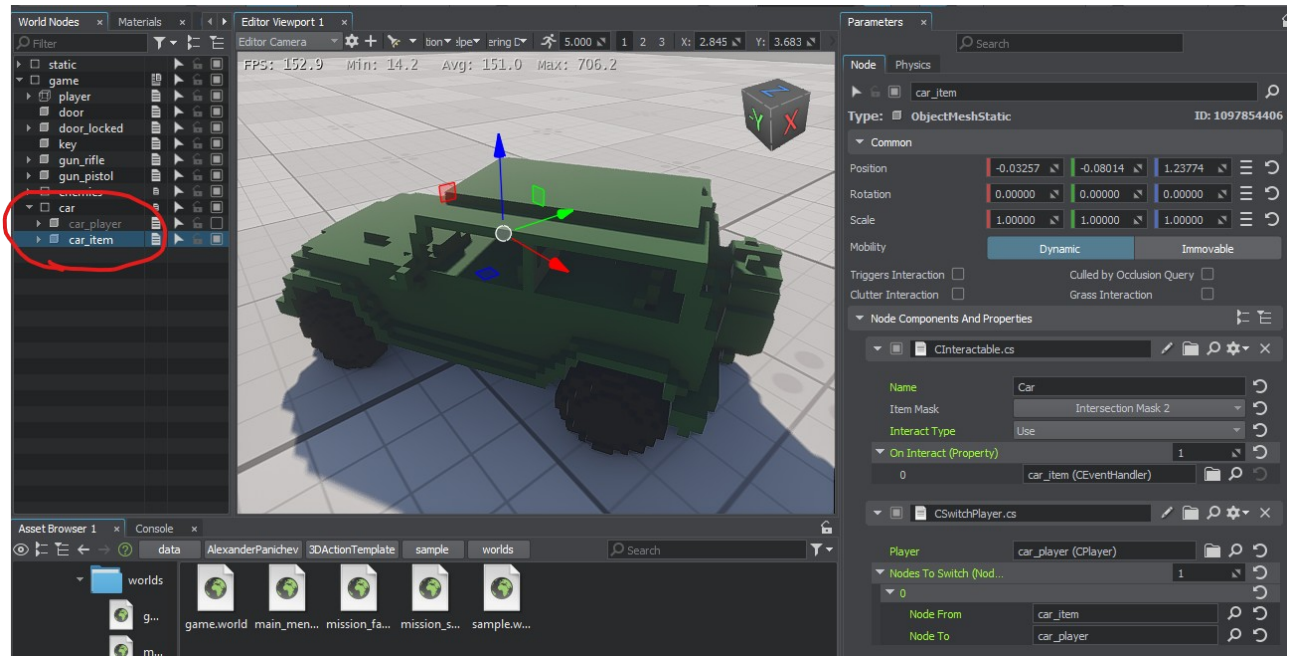

**Muzzle Flash** – flash effect node.
**Bullet File** – bullet node (must have **CBullet** component)
**Bullet Spawn Point** – position of node where bullets are emitted
**Shots Per Minute** – fire rate
**Bullet Spread** – inaccuracy
**Recoil Force** – recoil (animation only)
**Return Speed** – animation of returning after recoil

To deal damage, enemies must have the **CHealth** component:



# How Vehicles Work

Vehicles are alternate player types you can switch to. In the scene:




There are two nodes:

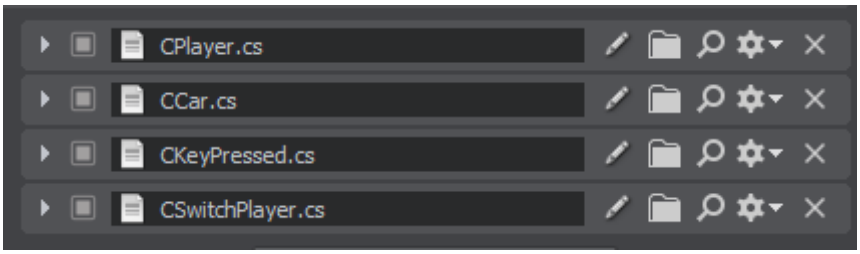
**car_item** – the car as an "item". Has two components:

**CInteractable** – allows “entering” the car.

**CSwitchPlayer** – switches control: disables current player, activates car_player, and hides car_item node.

**car_player** – the car as a "player". Active while driving. Has four components:

**CPlayer** – defines the car as a player (can it interact, where is the camera, etc.)




**CCar** – vehicle stats and controls. Like FirstPersonController but for cars.

**CKeyPressed** – fires OnPressed when a key is hit (e.g., F to exit). Calls Activate() of next component: CSwitchPlayer.

**CSwitchPlayer** - toggles back to car_item and restores player on exit.