# RPG System

The RPG system is a generic system to be used as a base for RPG games in Unity. Characters have health, energy, and a database of other stats such as speed, jumping and damage resistance to various damage types. These stats can be buffed or debuffed, and the buffs and debuffs appear as both particles on the character models and icons on their HUDs.

Each Character has a number of Powers, which can cause damage and buff/debuff effects to themselves or others. Cosmetic effects, damage types, energy cost and cooldowns are all set on scriptable objects for the Powers, and attached to Character components in the Inspector without any special code required.

Player powers can be activated via the keyboard and are displayed on a HUD.

AI powers are used automatically via a built-in utility AI system.

The Standard Assets Third Person Character Controller systems are used as a base for the animated characters.

## Coding Conventions and Folder Structure

The following folders are used throughout development

RPG – contains all scripts. All classes are in the RPG namespace

Example Buffs – stores all Buff and Power scriptable objects

Example Particles – all the particle system prefabs. Those intended to loop indefinitely to represent a buff are prefaced with “Buff\_” eg Buff\_Sparkles. Those intended to represent an instantaneous effect are prefaced with “Effect\_” eg Effect\_Flash.

Prefabs – character prefabs and other general purpose prefabs such as HUDs and projectiles are kept in here.

## Scene setup

In order to run, an instance of the **RPGSettings** object must be present. It should be set up on the main HUD canvas if you plan to have an automatic health bar placed on every Character for you. In the test map this is on the PlayerHUD canvas.

If you want floating damage numbers to appear in game, you’ll need an **ObjectPool** for the floating numbers, and to reference this via the Canvas.

While no persistent data is kept between levels here, you may want to set these up as persistent between scenes with DontDestroyOnLoad

Each Character can be represented using the Standard Assets Third Person character prefabs with a Character object attached to describe their RPG stats, and a Character Particles script to display buff effects on them. You can add the following components.

Player characters – **PlayerPowers** gives you power activation via the 1 to 9 keys. **TabTarget** gives you the ability to select targets via left mouse clicks or the TAB key.

AI characters – **NPCPowers** is a utility AI system for activating the powers placed on the Character.

## System Architecture Overview

The main classes in the RPG system are described below.

### Character

A Character is a basic character with health, energy, a database of named attributes, and a list of powers available for them to use. The Character also maintains a list of **Status** effects currently applied to them.

Each Character can be set up with a list of available **Powers** for them to use, either as a Player controlled character or an AI. This list can be changed at runtime.

### Status

An abstract ScriptableObject derived class, describing an effect that can be applied to a Character. The following subclasses can be instantiated. Active buffs on a character display an Icon on the Character’s HUD and particles on the Character in game.

The following subclasses can be created.

**Buff** – adds a positive or negative modifier to a named stat on a character, eg increases run speed, or reduced resistance to a particular set of damage types.

**DamageOverTime** – applies a number of damage ticks at regular intervals until it expires.

**Explosion** – an instantaneous effect (so no icon or particles) that triggers a **PowerArea** around the character.

### Power

An abstract ScriptableObject derived class, describing a power or spell that a character can use. This is where damage type, damage range, energy cost and cooldown and a list of **Status** effects applied by the power are defined.

The following subclasses can be created

**PowerDirect –**  a power that affects a single target. This can be the player themselves. Specifying a beam material in here will create a visible beam between the character and target when the power is used.

**PowerArea –** a power that affects all characters around the target within its radius

**PowerProjectile** – a power that creates a projectile that will deliver the power’s effect on any suitable target it hits. (This requires a projectile prefab be set up in the **RPGSettings** object). The Projectile script moves the projectile forwards using pseudophysics rather than a Rigidbody, and uses a small trigger collider to detect contact with the target.

### CharacterHUD

A master script that keeps a HUD to show the current state of a particular character.

It can display the following items:

**Portrait** – shows a portrait image for the character in any child called “Portrait”

**Name** – a child text called “name” will display the character’s name.

**Status icons** – will clone a member it finds called “Icon” for each Status on the character, and update its child “Counter” to show how many stacks of each buff are present.

**Health, Energy etc**. – see CharacterHUDMeter below.

The target Character can be set in the editor or switched at runtime, for example to show the character targeted by the player. Each CHaracterHUD can specify another CharcterHUD to show the Character’s target. If a CharacterHUD shows a null Character, it disables itself entirely.

### CharacterHUDMeter

A script that resizes an Image based on the named variable in the target character’s database. This is used for health and energy bars, but can also be used for any additional named properties you may wish to display. The example has an extra meter for displaying charge for blocks and maintains on the HUD.

### CharacterParticles

Responsible for spawning and cleaning up particle systems that play on a character to show active buff effects on them.

### RPGSettings

A singleton object that contains global information required by the system.

The OverheadHUD to instantiate for every character (by default set up to just show health and energy)

An ObjectPool for floating damage numbers to spawn from

The standard projectile prefab for projectile powers

The array of standard colours to use for tinting power effects. These are displayed as an array in the inspector. Their indices should match up with the **ColorCode** enumeration. The ColorCode enumeration is used by powers and buffs to provide a consistent visual look across the game, e.g. all powers tagged as Cold use the same blue hue. The final member of the enumeration is Custom, which allows the designer/user to specify a custom colour for a particular power.

If you want to add a new colour code, e.g. a dark red called “Demonic”, just add that to the end of the Colorcode enumeration before Custom and add an extra color in the data on the RPGSettings object.

## Character Animation

The RPGAnimations Animation Controller is a copy of the Standard Assets one with a few extra nodes added. Each node has a single transition back to the grounded state with one condition, that the ‘release’ Boolean must be true. This allows the scripts to get the character to hold the last frame of their animation while maintaining or charging a power, and then release it when its’d one.

The animations are triggered when the character uses a power by calling Animator.Play with the name of the node. The different animations available are stored in the **Animation** enumeration in the base **Power** class. The names of these enumeration members must match the names of the Animation controller nodes for this to work.

## Power and Particle Emanation points

Each power specifies where it comes from and where it targets, using a **BodyPart** enumeration in **Character.** e.g. a Blinding debuff plays a black cloud around the character’s head. A speed boost plays sparkles on their feet. A heat vision beam attack goes from the user’s head to the target’s root node between their feet.

These bodypart indexes are converted to a transform via some public Transform members and the function Character.GetBodyPart(). Currently, transforms for the head, hands etc have to be assigned in the editor for each character prefab manually.

## Damage Indicators

When a Character takes damage or is healed, number appear in 3D space to indicate the damage. The Numbers prefab uses a TextMesh to draw the numbers, and has a NumberFloater script for upward movement and fading out. After fading completely the GameObject is made inactive.

These work with a ObjectPool that instantiates a fixed number of copies upfront, and recycles them as required when a Character takes damage.

The Activate function positions the object, turns it on, and colours the text red for damage, green for heals.

## Damage Types and Stats

A number of damage types are specified in the RPGSettings enumeration **DamageType**. This includes basic types such as Crushing, Piercing and Magic, which are all powers of 2. It also has composite types such as Phsyical, which incorporates Crushing, Piercing and Toxic by bitwise OR-ing the basic types that make up the group. This allows the user to specify buffs with “resistance to physical damage” or “resistance to All damage” far more easily than having to fill in multiple identical entrties.

Extra composite damage types can be added by adding members to the DamageType enum, e.g.

Demonic = Fire|Toxic|Magic,

The Character’s database stores Resistance, Defence and Damage boosts by damage types. Each of these expand into a number of different stats such as ‘CrushingRes’ for resistance to crushing damage, and ‘CrushingDam’ for a boost to crushing damage, and so on.

Buff.Apply processes buffs with these StatNames in a special way, adding modifiers to any appropriate damage type based on the buff’s damage type.

Other **StatName**s can be added and then used in whatever code is appropriate, but Res, Dam and Def must remain the first three entries in this list.

For example, the Recharge stat is used to affect power cooldowns like so:

// update cooldowns

for (int i = 0; i < coolDowns.Length; i++)

{

if (coolDowns[i] > 0)

{

coolDowns[i] -= Time.deltaTime \* GetFactor(RPGSettings.StatName.Recharge);

if (coolDowns[i] < 0)

coolDowns[i] = 0;

}

}