Table of Contents

Hypercasual Runner Starter Kit User Manual

Simple and extensible package that allows any developer to create beautiful runner games whether he/she is an experienced or beginner.

You can find the package at Unity Asset Store

I highly recommend you to look over the demo scenes for learning purposes. Those example games created for you to give a sense of what is possible.

I also highly recommend that you come to our **Discord Server** for learning from others, chatting with developers or asking questions!

Getting Started

Scripts are documented. You can find them inside the scripts or via API references in here. Every gameObject that player controls have a Player component. It acts as a manager (composition root) for things like locomotion, or population managers. It listens inputs, enables or disables components, spawns things in startup. So you will see it in every example game.

RunnerMover is also a component that works really well for nearly every Runner game, so every example has one with different parameters.

Some kind of Monitor like PopulationCountMonitor will exist on every example too.

Core principles

One of the core concepts of this package revolves around PopulationManagers, GenericModifiers and Modifiables. PopulationManagers instantiates PopulatedEntities. GenericModifiers gets the required components from instantiated PopulatedEntities. For example, ProjectileShooterModifier gets ProjectileShooterModifiable from PopulatedEntities.

When GenericModifiers gets required Modifiables, they act on them. They can transform all of the Modifiables, level up them, increase or decrease scale. It signals them to shoot projectiles, trigger something continously, or set their animator states.

This way, you can create stacking games or crowd populating games, one of the most popular genre of hypercasual runners!

Master Of Counts

> **NOTE**
>
> For making crowd populating games like Count Masters, look at the example in `Demos/_Games/MasterOfCounts/MoC_Gameplay.scene`. You will see a Player object with all the necessary components on it. Crucial components for it to work:

- **CrowdManager**: It controls the crowd, organizes them, allow them to battle.
- **PopulationCountMonitor**: It updates the text if something has changed in CrowdManager.
- **AnimationModifier**: It uses regular Animation instead of Animator because of the performance reasons.

Gates uses **Collectable** component with **PopulationEffect** component, so together they can increase crowd when Player touches it. It also uses **CollectableChoice** component for disabling other Collectables when any one of them is picked up.

JumpTrigger is used for (as you might guess) jumping PopulatedEntities. You can customize the behaviour via parameters.

BattleAreaTrigger instantiates BattleArea and starts battling of CrowdManagers. It also has it's own CrowdManager.

BattleArea is a prefab that BattleAreaTrigger spawns when battling starts. You can customize movement speed of battling entities in there.

PopulatedEntityObstacle type is selected to Abyss. Abyss allows touched PopulatedEntities to fall.

For understanding what every parameter does, please read the tooltips and comments.

Evolution Of Crowds

> **NOTE**
>
> This is another crowd populating game which is similar to the County Masters. You can find the example in `Demos/_Games/EvolutionOfCrowds/EoC_Gameplay.scene`.

Characters go throught gates and they evolve into better characters. They attack more with different projectiles. Characters face enemies that can kill our characters. They also have a health so we can kill those.

Player gameObject is similar to the one from Master Of Counts game. We are now using **TransformationLevelMonitor** instead of PopulationCountMonitor. One key difference is that now we have a component TransformationModifier. It TrasnformationModifiables, which can mean activating/deactivating child gameObjects of one gameObject, or it can mean custom thing that you can bind with UnityEvent. When you disable Should Change Game Objects, you can use UnityEvent. However in this context, we are changing gameObjects.

Other difference is we also have ProjctileShooterModifier component. Remember that PopulatedEntities need to have corresponding Modifiable component in order to work! ProjectileShooterModifiable listens Transformation events, so if one of our child gameObject changes, it grabs that gameObject and uses it's ProjectileShooter. One core difference between this Modifiable with the other ones is that this requires you to assign ProjectileShooter to every child gameObject of PopulatedEntity. This way every child can shoot something different, which is what you would want!

Gates now uses Collectable with **TransformationEffect** which manipulates transformations. If **useDirectTransformation** is enabled, it directly increases transformation level, thus shows immediate effect on the gameObject it touches. If however useDirectTransformation is disabled, then it adds Experience which is a generic term for telling when experience passes some threshold, it levels up, instead of directly increasing. If useDirectTransformation is disabled, you also need to set it on TransformationModifier. Then it will require you to add a **IntThreshold**, which is a experience threshold that you can set for levelling up.

DamageableEntities have a **Damageable** component, **DamageableMonitor** which listens Damageable and shows the current health. It also has a **BlinkFeedback** which Damageable uses when takes hit. It's also PopulatedEntityObstacle with **Kill** type with hasLimitedLife enabled. Which means it can't kill every PopulatedEntities you have, it will die after that many entity it kills.

For understanding what every parameter does, please read the tooltips and comments.

Evolution Of Cars

> **NOTE**
>
> You can find the example in `Demos/_Games/EvolutionOfCars/EoCars_Gameplay.scene`.

This is a game where we control a vehicle. There are gates which when we pass through, they evolve vehicle. The more evolved the vehicle is, the faster it goes. There are obstacles that kills vehicle when vehicle touches those.

Player gameObject is similar to the one from Evolution of Crowds game. We don't need to use projectile shooting so we have removed ProjctileShooterModifier component. We only have one PopulatedEntity which is fine, it's a racing game! But take a closer look at the child gameObjects. They have a component **MoveSpeedDefinition** which changes RunnerMover speed when gameObject is enabled. This way every vehicle can have different move speeds!

Gates now uses experience instead of direct transformation. TransformationModifier component has a VehiclesThreshold which holds values for transitioning to the next level. We also have small lightning collectables which acts the same as gates, but they don't have any CollectableChoice parent object. We have Tenderizers with ObstacleType Kill, they are basically threat we shouldn't touch. Obstacle Abyss is also threat which kills our vehicle.

For understanding what every parameter does, please read the tooltips and comments.

Run Of Destiny

> **NOTE**
>
> You can find the example in `Demos/_Games/RunOfDestiny/RoD_Gameplay.scene`.

This is a game where we control a character, it walks on the road and there are gates and collectable things on the road. Collectable things are in two category, one is good one is bad. We try to collect one type who serves what we aim for. Like if you want to become an angel, you need to collect good things (or green).

This game is quite similar with the Evolution Of Cars game. It shares most of the fundamental mechanics.

Gates and heart/diamond shaped collectables gives experience points.

For understanding what every parameter does, please read the tooltips and code comments.

Stack Of Coffees

> **NOTE**
>
> You can find the example in `Demos/_Games/StackOfCoffees/SoC_Gameplay.scene`.

This is another side of the pack! Player gameObject uses **StackManager** instead of CrowdManager this time. StackManager acts very similar to the CrowdManager but instead of randomly pulling objects to the center, StackManager aligns them. There are tons of similar games which uses this mechanic.

Other than StackManager, it uses good old TransformationModifier. In other examples, PopulatedEntities spawned as a child of the Player gameObject, but this time we don't want that because StackManager will align the gameObject by manipulating their Transforms, so they should be unparented.

> **IMPORTANT**
>
> Thus, if you want to see smooth movement of PopulatedEntities, don't forget to disable shouldSpawnEntitiesAsChild.

We have **TransformationSingle** component which transforms PopulatedEntities one by one instead of all of them at once. This is useful because in a lot of stacking games this is the behaviour you want to have!

For understanding what every parameter does, please read the tooltips and comments.

Modifying Populated Entities

You can define custom behaviours that affects PopulatedEntities in two ways.

1. Modifying all of them at once
2. Modifying one at a time

Below you can see the corresponding solutions for the use cases:

1. Extending GenericModifier, you can create new type T that allows you to modify every PopulatedEntity. Projectile shooting, playing animations or transforming objects uses the same logic behind the scene. So feel free to add your own Modifier - Modifiable pair and modify PopulatedEntities!
2. You can directly affect PopulatedEntities one at a time instead of everyone at the same time. It's more straightforward, you can see from the example:

```
/// <summary>
/// Transforms TransformationModifiable one by one, just like cup filling coffee machines in Coffee Stack game
/// </summary>
public class TransformationSingle : MonoBehaviour
{
    [SerializeField] int _levelToSet = 1;

    void OnTriggerEnter(Collider other)
    {
        if (other.TryGetComponent(out TransformationModifiable modifiable))
        {
            modifiable.SetLevel(_levelToSet);
        }
    }
}
```