Back Story

Hi and thank you for purchasing the Procedural Car Builder tool for Unity.

This tool is an extension of the “Simple Mesh Generator” tool, and is already included in this package. I advise you to read up on the Mesh generator’s documentation as well.

The main objective of this tool is to allow you to configure one or multiple cars, to be used in your project alongside your existing assets. Please take a look at the provided Demos !

Cars can be generated at runtime, which can be ideal if you aren’t sure beforehand what car is needed. Thus allowing you to generate it with the necessary modifications.

Cars can be saved as Prefabs, allowing you to create a library of usable cars.

Class Functionality Details

CarGenerator.cs
Here you can generate a car by providing it the necessary data, which can be generated from a settings asset.

The *“createNewCar”* parameter controls whether or not the generator will re-use/modify the last car it generated or create an entirely new mesh. When generating a multitude of cars for your app/project it will likely have to be set to *true*. But when modifying a single car, you wouldn’t want to keep generating new meshes which fill up your memory.

CarSettings.cs
This is a Scriptable object holding all the specificatications on how you want your car to look like. It holds a range of minimum and maximum values, which allow to define a range for (nearly) every variable.

By calling **GenerateData()** you will create the data structure which goes into the aforementioned **CarGenerator.cs**.

The *“safeMode"* parameter controls whether or not to automatically perform a **SanityCheckData()** after processing all data.

The *"tweaking"* parameter will control whether or not to rechoose a random result when an array of options are provided. Example: If there are 3 wheel types to choose from in the DataSet, you don't want to randomly get one of them each frame when making continuous edits.

CarData.cs
This is the data holder which is the “bridge” between the 2 earlier mentioned classes, it holds a bunch of data to facilitate the car generation. It includes the **SanityCheckData()** function. This function will modify some of the data when inconsistencies and/or impossibilities are found, while this does cover every single possibility, it takes care of the biggest ones.

Example: If you define the wheel radius to be 2 and at the same time make the body of the car be only 1 meter high, it would mean that the wheels stick out above the frame, this is considered “not possible” thus one or multiple parameters are adjusted to ensure a sensible car gets created.

CarPrefabSaver.cs
This class allows you to create a prefab of a car.

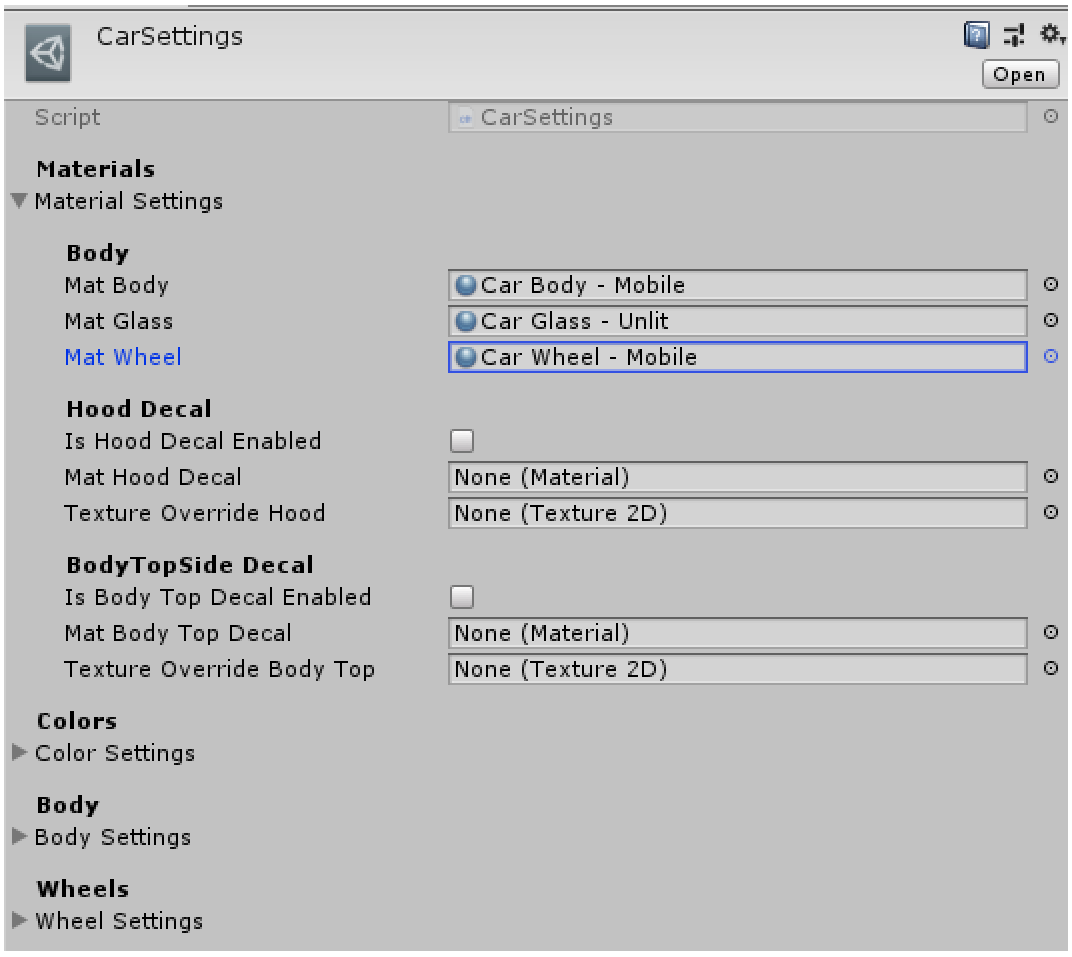
CarPartReferences.cs
This script is automatically attached to any car generated and holds a bunch of public references to expose modifiable / moveable sections of the car including: Wheels, trunk, bonnet, hood, lights, roof, hood ornament, etc.

Car building Tutorial

Let's go through the steps of creating a car from scratch, do note that the Demo files cover all of these steps as well.

1. **Creating a settings file:** In the top most bar of Unity, click on "Assets" > "Create". This will open a list of assets to create for your project (like Material, AudioMixer, etc), go to "CarBuilder" and click on "CarSettings". This will add a CarSettings.Assets file in your project under the folder currently selected.

2. **Modify settings part 1:** Now we can start defining what our car should look like, select the CarSettings file and a bunch of settings will populate the Inspector window. Let's start with defining what Materials we want our car to be rendered by, for now assign the already included materials:



Script: CarSettings

Materials
▼ Material Settings

Body

Mat Body	Car Body - Mobile
Mat Glass	Car Glass - Unlit
Mat Wheel	Car Wheel - Mobile

Hood Decal

Is Hood Decal Enabled	☐
Mat Hood Decal	None (Material)
Texture Override Hood	None (Texture 2D)

BodyTopSide Decal

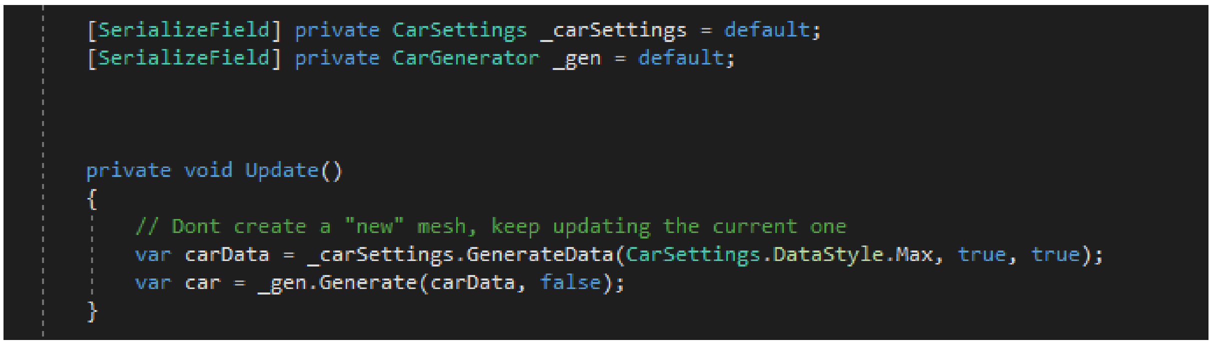
Is Body Top Decal Enabled	☐
Mat Body Top Decal	None (Material)
Texture Override Body Top	None (Texture 2D)

Colors
▶ Color Settings

Body
▶ Body Settings

Wheels
▶ Wheel Settings

3. **Generating the car:** Create a new scene > add a Game object > add the CarGenerator component to it. Create a new script which will need to hold 2 references, one to the generator and one to the settings asset. We will create a Car every frame with the *"createNewCar"* variable set to *false*. Once completed press "Play" in your unity editor. Example:



```
[SerializeField] private CarSettings _carSettings = default;
[SerializeField] private CarGenerator _gen = default;



private void Update()
{
    // Dont create a "new" mesh, keep updating the current one
    var carData = _carSettings.GenerateData(CarSettings.DataStyle.Max, true, true);
    var car = _gen.Generate(carData, false);
}
```

4. **Dude, Where's My Car?:** While in Play mode you should now see a car in the center of your scene, but it doesn't look like much yet. No wheels, everything is red and the proportions might not be favourable. Also there will be a bunch of warnings as not all necessary / chosen fields have been assigned yet, this is okay. It's time to make it look a bit nicer.

   Open the Color tab, define 4 colors to start off with, (mentally) define these as Body, Glass, Metal and Tire. Then in the Color Settings of each car part you can choose which color to use.

   Next let's add some wheels and a wing mirror. Go to the Wheel Settings tab, there you will see the first parameter *"Wheel Cap Style"*, its an empty array, change the size to 1 and add any of the readily available wheel assets. ***Note*** *if you do not see the wheels appearing in the scene, stop play mode and start it again. When generating a car the last parameter "tweaking" will prevent some settings from being manipulated after the first generation. Thus if it's enabled you will need to exit/enter play mode again.*

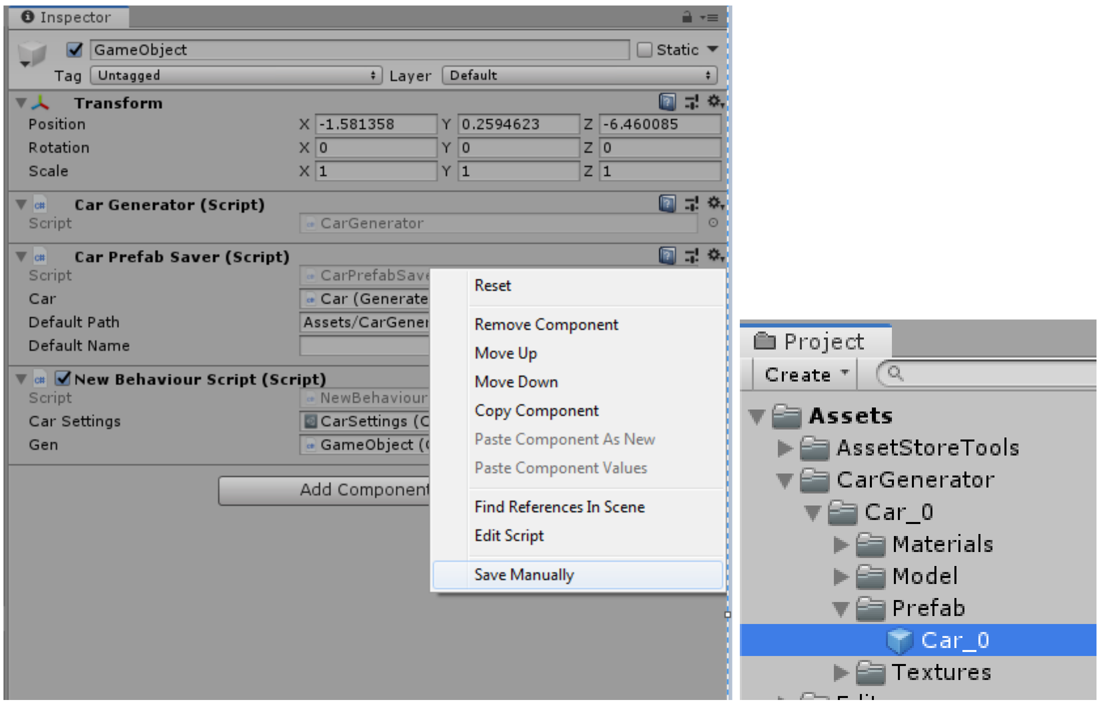
   The wing mirror works the same way, go to the Wing Mirror Settings tab and add an asset to the *"Shapes"* array. You should see them appear on the car right away or after re-entering play mode (see previous note).

5. **Modify settings part 2:** As the car generator keeps recreating the car, you can keep modifying the CarSettings asset until you have the car you want. These setting changes will be automatically saved by Unity (even during play mode as it's an asset!).

6. **Using the car in your project:** You can set up your own script to generate multiple cars at runtime based on one or several different Car Setting assets. These cars can even be modified to comply with specific assets already in your project (take a look at Demo2 for an specific example).

   Alternatively you could save the generated car as a prefab, making it a regular asset in your project. To do so add the “CarPrefabSaver” component to your gameobject as well. Pause Unity (If you are generating the car continuously, like we have done so far in an Update loop) and drag and drop the generated car from your hierarchy view into the “Car” parameter. Click the Gear Icon on the Prefab saver and select “Save”. The car is now saved as a Prefab under the specified path.

   *(**Note**: Occasionally Unity throws errors relating to: <u>UnityEngine.UIElements.UIRAtlasManager.BeginBlit.</u> I’m trying to get this fixed as of writing this, but luckily haven’t noticed any issues / impact to the prefabs yet)*



Inspector
GameObject — Static
Tag: Untagged — Layer: Default

Transform
Position: X -1.581358, Y 0.2594623, Z -6.460085
Rotation: X 0, Y 0, Z 0
Scale: X 1, Y 1, Z 1

Car Generator (Script)
Script: CarGenerator

Car Prefab Saver (Script)
Script: CarPrefabSave
Car: Car (Generate
Default Path: Assets/CarGener
Default Name:

New Behaviour Script (Script)
Script: NewBehaviour
Car Settings: CarSettings (C
Gen: GameObject (

These steps should cover the basics on how to generate cars based on a CarSettings asset. Once you are happy with your settings file you can easily create some variations based on it by either duplicating the asset and modifying these or tweak both the Min and Max values and letting the generator randomly choose between them. The Demo CarSetting assets should help you get started.

Wheels

When following the tutorial we added a *WheelStyle* asset to our settings asset. What is this Style asset and how does it work?

Wheels give a car a lot of character, signify its primary function and can hint at it's age. With endless possibilities as to how wheels and their caps can look like, I've decided to leave the design of the wheel to the end user, with 3 examples provided.

By extending the CarPartScriptable.cs you can define your own ScriptableObject scripts to generate a wheel (or extend it to even create other car parts).

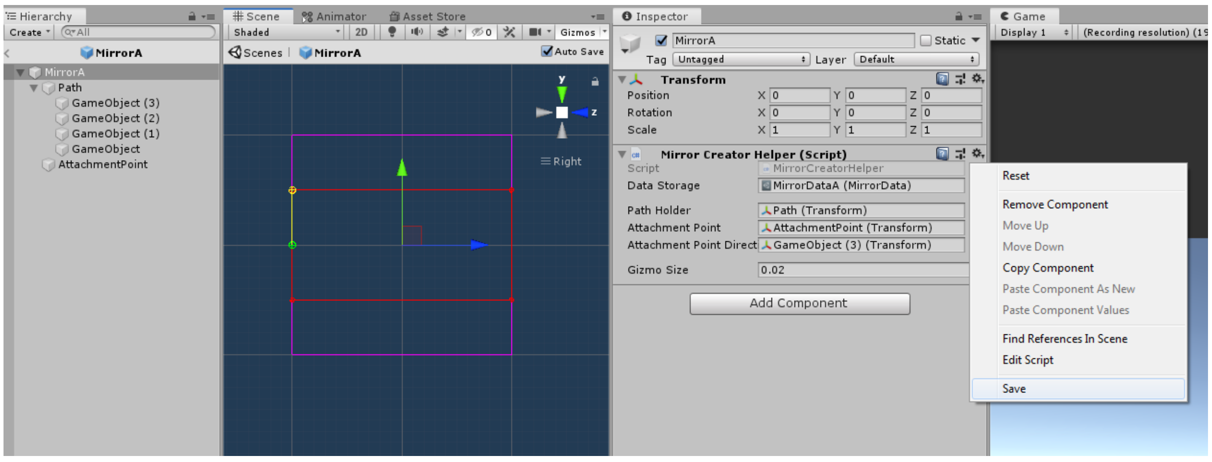
The simplest example is "WheelCap_Round" which based on the given input (radius, etc) creates a mesh and passes it back to the car generator for inclusion.

"WheelCap_Spoked" is slightly more complex which also exposes some settings unique to its design. This scriptable object might lead you to generate multiple asset files with different values assigned to its Design specific parameters.

Shapes

Besides the wheels there are other body parts whose design could be left open to the end user. Examples of these are the wheel fenders ridge, nose ridge and wing mirror. These car parts are defined in separate Assets which hold all the relevant details. Such shapes often need visual representation to be properly defined thus I would like to demonstrate how one would go about defining these:

When opening the "MirrorA" prefab you will be presented with a prefab which only holds a set of linked transforms, defining a shape/path within the purple outlined bounds. The prefab also contains a script with a reference to the intended MirrorData Asset file, when right clicking the gear icon on the script you can save the shape data into the asset. This asset is then ready to be used by the car generator, allowing you to customize the designs of these parts.



MirrorA
- MirrorA
  - Path
    - GameObject (3)
    - GameObject (2)
    - GameObject (1)
    - GameObject
  - AttachmentPoint

MirrorA Static
Tag Untagged Layer Default

Transform

Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Mirror Creator Helper (Script)

Script	MirrorCreatorHelper
Data Storage	MirrorDataA (MirrorData)
Path Holder	Path (Transform)
Attachment Point	AttachmentPoint (Transform)
Attachment Point Direct	GameObject (3) (Transform)
Gizmo Size	0.02

Materials

In the Material section of the car settings asset, you can define the following materials.

Body: Material applied to the entire car frame
Glass: Material applied to all “glass” sections like: the windows and the wing mirror
Wheel: Material applied to the entire wheel

Hood Decal: The intention of this Material is to be applied to the hood area of the car and be rendered on top of the body. By using the provided *Carbuilder/Decal* Shader the material should make use of UV channel *Two* to properly map the decal based on the chosen Fitting & Anchoring modes.

Lastly the RenderQueue needs to be higher than Geometry (2000), but I recommend putting it at Transparent + 10 (3010). To minimize potential conflicts with other materials. See provided Demo material.

BodyTop Decal: The intention of this Material is to be applied to the top area of the Car (Body, Roof, Hood) and be rendered on top of the body. By using the provided *Carbuilder/Decal* Shader the material should make use of the UV channel *One* to properly map the decal based on the chosen Fitting & Anchoring modes.

Lastly the RenderQueue needs to be higher than Geometry (2000), but I recommend putting it at Transparent + 5 (3005). To minimize potential conflicts with other materials. See provided Demo material.

Other

Decal Images provided by : https://pixabay.com/

3D Fruit Assets Provided by Unity:
https://assetstore.unity.com/packages/3d/food-pack-3d-microgames-add-ons-163295

Notes

If you have any questions and/or suggestions please contact me either through my email listed on the asset store or send me a message to my Unity account.

- The Demo folder can be removed without loss of functionality, but its recommended to keep it around for a while as a source of reference

Author

Glenn Korver