AVK | Assault Vehicle Kit

Version 1.1

User Documentation

Updates:

Version 1.1

- Introduced a modular system of camera, steering, and aiming controllers, along with demo UI elements to choose and display the controllers within the demo scene.
    - Camera controllers:
        - Orbit Cameras (World and Vehicle Up with optional offset)
        - Follow Cameras (World and Vehicle Up)
        - Cockpit Cameras (Free Look and Fixed)
        - Birds Eye Cameras (Follow and Fixed)
        - Top Down Cameras (Follow and Fixed)
        - CCTV Camera
    - Steering controllers:
        - Camera Steering (Camera view vector controls steering direction)
        - Manual Steering (Vehicle and Camera Relative)
    - Aiming controllers:
        - Camera Aiming (Camera view vector controls aiming point)
        - Manual Aiming (Screen Cursor and Camera Relative)
- Added enable/disable options for Projectiles to control telemetry and/or to handle collisions. This allows external components to handle those aspects of the projectile while still using the Projectile's damage and other settings.
- Weapons can now instantiate generic GameObjects instead of requiring Projectiles.
- System level events now propagated via the Events class.
- Misc code cleanup and tweaks.

Version 1.0 – Initial release.

Thank you for purchasing AVK | Assault Vehicle Kit!

AVK is a modular system of interworking components providing a clean and simple template for bringing assault vehicle fun to your projects. The system is ready to use as is, but can be easily extended with your own custom components – be they controllers, vehicles, turrets, weapons, projectiles, impact prefabs and more. Drop your custom components into the scene and watch them play along without changes to the rest of the system.

AVK can support vehicles of any type, not just wheeled vehicles – multi to single wheeled vehicles, hover craft, flying vehicles, amphibious vehicles – you name it. The provided vehicles were built with an emphasis on fun, not realism, but AVK can also be integrated with other vehicle systems, including more realistic ones, if so desired. As long as the vehicle can be operated with input such as move direction or steer angle, throttle and braking, it can be integrated with AVK.

Features:

- Clean and well-commented C# source code.
- Mouse, keyboard, and Xbox 360 controller input supported.
- **NEW** modular and extensible controller system for the camera, steering, and aiming subsystems that enable you to easily tailor the driving & aiming/shooting mechanic of your game.
- Modular system of vehicles, turrets, weapons, projectiles, and impact prefabs. Mix and match the components of the system and watch them all play along (see playable demo).

- Fully playable demo and mini-game. Drive around and blow stuff up to earn experience.
- Damage text particles.
- Enemy floating health bar.
- Entity health and shield regeneration.
- Player status HUD showing shield, health and experience.
- Enemy AI to drive vehicles around and shoot at nearby Entities.
- Simple waypoint system for path following AI.
- Player re-spawn circular timer UI.
- Experience gained UI effect.
- Many more extras, such as: billboards, fading projectors to minimize projectors in the scene, auto un-parented particles for realistic projectile effects.

Demo Scene:

The provided demo scene, “DesertDemo”, is located in the AssaultVehicleKit/Scenes/ folder. This demo provides a complete scene and mini-game allowing you to drive around and shoot at enemy vehicles for experience - of course, they will shoot back!

The demo scene should exemplify all aspects and features of AVK, including the modular nature of the components. For instance, the enemy AI component is exactly the same for all vehicles and turrets, regardless of vehicle type (Assault Buggy or RollerBot), and can drive around and shoot no matter what vehicle you place them on. The player can even swap out vehicles on the fly in the middle of the action

– hit the [Tab] key – without any change to the driving and aiming components.

We hope you enjoy the provided demo scene and mini-game.

Using the System:

The following describes what is included in the AVK system and how to use the provided components.

Vehicles:

All vehicles in AVK derive from a base **Vehicle** component type. This allows other components that need to interface with a vehicle (player driver, enemy AI, etc.) to control any vehicle in a generic way, regardless of the specifics of each vehicle. The components that control a vehicle, like the PlayerDriverCameraRig, will have a "Vehicle" reference where any vehicle can be placed into.

There are two vehicles provided in AVK currently, the "AssaultBuggy" and "RollerBot":

The AssaultBuggy is a wheeled vehicle with augmented "fun" features like increased stability (corners like it's on rails!), orientation correction in air and flipping back over, speed boost, hand brake power slide, and more. The AssaultBuggy accomplishes this by being built on top of

several layers of Vehicle types, each providing more specific vehicle features. Looking at the AssaultBuggyPlayer prefab in the inspector shows these layers and the parameters associated with each:

Vehicle (base type):

Orbit camera parameters (specific to each vehicle) are part of the base Vehicle type and help control the player orbit camera.

Wheeled Vehicle:

The wheeled vehicle parameters provide basic control for generic wheeled vehicles, such as max turn angle and turn speed, max forward and reverse speed, max and min engine RPM, max torque and torque speed curve to simulate engine limits. Gear ratios and auto gear switching as well.

Augmented Wheeled Vehicle:

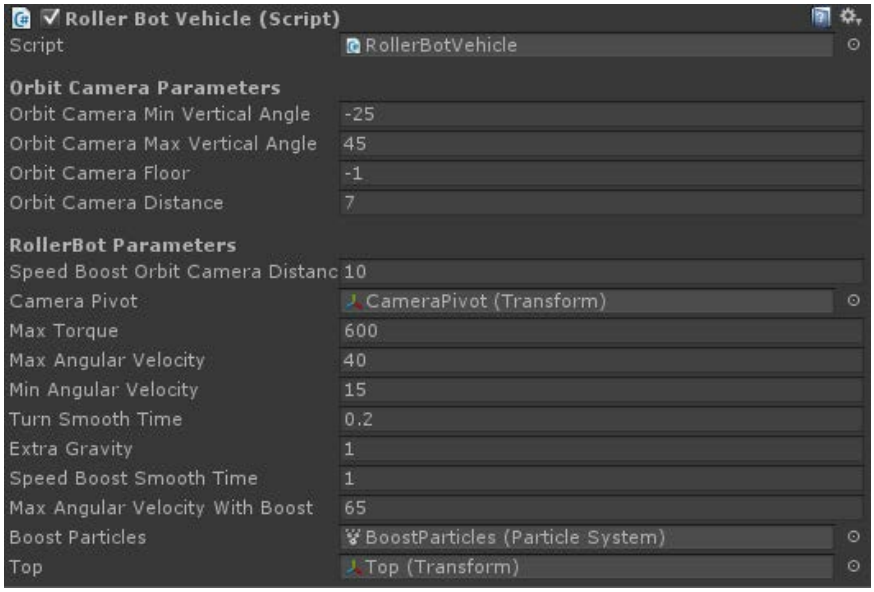
The augmented wheeled vehicle starts to add the “fun” factors into driving, including a down force for extra stability, speed boost parameters, max and min wheel sideways stiffness to facilitate fast driving and turning without flipping over, along with in-air and flipping torques to land back onto wheels.

Assault Buggy:

These parameters provide for the specifics of the Assault Buggy model itself, including the left and right boost thruster particle effects, steering wheel, auto braking when no throttle is provided, and braking/rear lights.

The RollerBot is not a wheeled vehicle and, as such, does not utilize any of the wheeled vehicle components. Instead, the RollerBot vehicle type just extends the base Vehicle type directly. The RollerBot is primarily a rigidbody sphere rolling around with applied torque.

But since the RollerBot is a Vehicle (derived from the base Vehicle type), it can be driven by any of the systems designed to control vehicles (player driver and enemy AI).



Vehicle (base type):

All vehicles implement the base Vehicle type, which includes the orbit camera parameters specific to each vehicle.

Roller Bot:

These parameters provide the specifics of the RollerBot, including the min and max angular velocity, turn smooth time, speed boost ramp up time and the boost particle effects.

Player Controllers:

All player controllers in AVK derive from a base **PlayerController** component type. There are three types of player controllers: camera, steering, and aiming. These three controller types together provide the full driving and aiming/shooting mechanic for the player and can be used to tailor the mechanics for your specific game.

Camera Controllers:

The camera controllers allow the player to either actively control the camera (as in orbit cameras), or passively control it by controlling the vehicle and the camera reacts accordingly (as in follow cameras). All camera controllers derive from a **CameraController** component type (which is itself derived from PlayerController, as are all controllers below).

Steering Controllers:

The steering controllers allow the player to steer the vehicle in various ways, either manually or by controlling the camera. All steering controllers derive from a **SteeringController** component type.

Aiming Controllers:

The aiming controllers allow the player to aim and shoot the turret(s) on the vehicle. All aiming controllers derive from the **AimingController** component type.

Turrets:

All turrets in AVK derive from a base **Turret** component type. This allows other components that need to interface with a turret (player aiming and firing, enemy AI, etc.) to control any turret in a generic way, regardless of the specifics of each turret. The components that control turrets will generally obtain references to them as children of the parent vehicle the turrets are attached to.

The provided turret examples just implement the specifics for each turret, like pivot speed and pitch aiming limits – but every turret at its base has the ability to aim at a target point and fire primary or secondary weapons.

Weapons:

The provided base **Weapon** component acts as a generic weapon that is attached to a Turret as either a primary or secondary weapon. The Turret sends firing commands to the Weapon when prompted by the user or AI. The Weapon component contains parameters to control the firing rate, firing accuracy (based on muzzle heat), muzzle velocity and kickback forces, and muzzle heat per shot and cool down rates. The Projectile prefab that is fired by the Weapon can be specified, along with the muzzle velocity and the max life duration of the projectile.

If more weapon features are required, the base Weapon component can be extended (derived from) to add the extra features, while any Turret will still be able to send firing commands as usual. The

LaserBeamWeapon is an example of this, which is a derived Weapon that displays a laser beam upon firing.

Projectiles:

The base **Projectile** component acts as a generic projectile that is fired from a Weapon (which is attached to a Turret). The Projectile specifies the min and max damage done by the projectile, the min and max critical damage and critical chance, the impact and explosion forces (if any), whether the prefab should impact immediately without traveling, and if any particles should be un-parented upon collision – like rocket exhaust which would look funny disappearing immediately. The generic Projectile is also capable of casting a Raycast per frame to manually check for collisions since some fast projectiles have issues with Terrain collision, even with Continuous Dynamic set on the Rigidbodies.

Each Projectile can also specify any number of impact prefabs that should be created upon collision. Each impact prefab has a filter for the valid layers, tags, component types and excluded component types of the collided object that the prefab can be instantiated for upon collision. You can specify zero to many prefabs per collision with this system.

Like the Weapon type, the base Projectile can be extended to provide extra features if needed, with the Weapons still able to fire them.

Impact Prefabs:

The impact prefabs are simply game object prefabs instantiated upon Projectiles colliding with an object. The impact prefab can be any type of game object, like decals, particles systems, etc. The impact prefab will be created by the Projectile and will be placed and oriented according to the impact point and normal, along with the Projectile's prefab parameters.

Entities:

The **Entity** component provides shield, health and experience for the player and enemies in the scene. A death prefab can be specified to instantiate on the Entities death as well. Every Entity in the scene stores information on what Entity has done damage to it and will award experience to the Entity that has done the most damage and is still alive when the Entity dies.

There is also an **EntityRegen** component that will regenerate either shield or health for an Entity at a given rate and reset time when attached to the same game object as an Entity.

Player (Entity, Input, Driver, and Re-spawner):

The **PlayerEntity** is a special Entity that will send out a "playerEntityStarted" event that other components listen to in order to find and attach to the player's Entity. There should only be one PlayerEntity in the scene at a time.

The **PlayerInput** component reads player input from the mouse and keyboard, or Xbox 360 controller, and turns them into player actions. The player actions are static members of the PlayerInput component and can be easily read by the rest of the system.

The **PlayerDriver** is the main driving and aiming system for the player. The PlayerDriverCameraRig prefab (which has a Main Camera as a child) reads the player actions and controls the orbit camera, along with sending commands to the player's vehicle and turrets.

The PlayerDriverCameraRig has a "Vehicle" reference that can be manually set (optional), but it will also attempt to find the player's vehicle by listening to the "playerEntityStarted" event and will attach to the player's vehicle that way.

The **PlayerRespawner** also listens to the "playerEntityStarted" event to find the PlayerEntity and will listen for the player Entity to die. Upon death it will start a re-spawn timer and re-instantiate a player prefab when the timer goes off.

AI:

There are a few AI based components in the system:

The **WaypointPath** represents a simple waypoint path that can be queried for each consecutive waypoint position or the nearest waypoint to a given position.

The **WaypointPathFollower** is a base component allowing for a path to be specified. The **AggressivePathFollowerAI** is a WaypointPathFollower and is the main component for all AI in the system. The AggressivePathFollowerAI is capable of controlling a Vehicle and multiple attached Turrets and will simply follow a specified path and will target and shoot at any other Entity within range.

The **WaypointAISpawner** spawns a given number of Entity prefabs along a specified path with a delay between spawns. This component listens for the death event of each Entity and will spawn a new one after the specified delay, keeping the total along the path constant.

UI:

The demo scene comes with **PlayerControllerDisplay** UI elements that will indicate the currently chosen player controllers being used to control the camera, vehicle, and turrets, along with simple guides of how to cycle between each type (C for Camera, R for Steering, T for Aiming). You can toggle the visibility of the entire controller display by hitting the M key.

The **Billboard** component provides billboarding behavior for any game object in the scene, keeping the object oriented towards the reference camera. The Billboard can provide 3 modes: Standard billboard (always oriented to the camera forward and up), vertical billboard (like trees), and horizontal billboard (ground effects).

CircularTimer is a timer UI used to show a re-spawn timer for the player upon death. The player will be re-spawned at the end of the timer.

TextParticle is a text particle UI component showing damage text particles in world space, falling off the Entity as damage is done.

DamageTextParticle is a component that is attached to an Entity game object and will listen for damage taken events. When damage is done, this component creates TextParticles with the provided damage details to be displayed.

EntityXPText is a text UI effect that shows experience gained for the player when an enemy is destroyed, fading upwards from the location on screen.

PlayerXPGainedUI listens for the experience gained event of the PlayerEntity and will instantiate an EntityXPText effect at the screen space location of the destroyed Entity.

HealthBar is used to show Entity health floating above the Entity. Usually accompanied by the Billboard component to always face the reference camera.

MenuController controls the help and settings menu.

PlayerStatusHUD attaches to the player by listening to the playerEntityStarted event and will show shield, health and experience for the player.

Extending the System:

The following describes how to extend the AVK system with your own custom components.

Vehicles:

All vehicles in AVK derive from the Vehicle base class (Vehicle.cs). The base class provides information about orbit camera parameters (unique to each vehicle depending on shape and size) and a single input property (Vehicle.input) where the vehicle processes driving input.

The VehicleInput structure (VehicleInput.cs) provides the means to communicate with a Vehicle. All members of the structure are nullable so that controllers can send only the information they need to. The Vehicle processes only the information passed to it each time. This could facilitate multiple controllers providing input to the vehicle without interfering with each other. Passing the input in a structure like this also allows for easy extensibility in that future additions can be added to the structure without breaking existing implementations, as long as existing members are not changed.

In order to define a custom Vehicle, simply derive from the Vehicle base class and implement the Vehicle.input property setter, where you will process VehicleInput parameters.

AVK derives several Vehicle types to provide the basis of some common vehicle types that can be used as a base and built upon:

The first is a SteerableVehicle (SteerableVehicle.cs) that defines vehicles that are steered along a horizontal “ground” plane, like cars or boats. The SteerableVehicle will convert the move direction from the VehicleInput into a target steer angle (relative to the forward direction of the vehicle). This target steer angle can then be used to turn a vehicle steer angle towards, perhaps with limits on the max steer angle and turn speeds. Forward and reverse throttle are also picked up at this level and used to inverse the steer angle when going in reverse.

The WheeledVehicle (WheeledVehicle.cs) is a SteerableVehicle that has wheels and provides for the basic aspects of such vehicles, such as: steering, power, braking and hand-braking wheels, engine RPMs and torque curves, gears and automatic gear switching.

The AugmentedWheeledVehicle (AugmentedWheeledVehicle.cs) adds certain “fun”, and unrealistic, aspects to WheeledVehicles. These features include things like a down force for increased stability, boost speeds, power hand-brake slides, etc.

To customize any of these Vehicle types simply create a Vehicle using them as a base and the rest of the components will be able to send your component VehicleInput commands as expected.

Player Controllers:

All player controllers in AVK derive from the PlayerController base class (PlayerController.cs). The base class provides a signature for both initializing (Initialize method) and exiting (Exit method) each controller. The initialize method is called just before the controller is first used while the exit method is called after the controller will no longer be used. The three derived controller types (camera, steering, and aiming) further provide unique update methods for their particular subsystem. A control references structure (ControlReferences.cs) is passed to each controller during use and contains important references, such as the camera and vehicle references that are needed by the controllers to process player input.

The CameraController (CameraController.cs) base class extends the PlayerController to provide an UpdateCamera method that is used by camera controllers to return a CameraInput structure (CameraInput.cs), which contains the information for controlling the camera.

In order to define a custom CameraController, derive from the CameraController base class and implement the Initialize and Exit methods (or leave them stubbed out if not required – see examples), along with the UpdateCamera method where you return a CameraInput structure which will be used to control the camera.

The SteeringController (SteeringController.cs) base class extends the PlayerController to provide an UpdateSteering method that is used by steering controllers to return a VehicleInput structure (VehicleInput.cs), which can be passed on to the Vehicle for processing.

In order to define a custom SteeringController, derive from the SteeringController base class and implement the Initialize and Exit methods (or leave them stubbed out if not required – see examples), along with the UpdateSteering method where you return a VehicleInput structure.

The AimingController (AimingController.cs) base class extends the PlayerController to provide an UpdateAiming method that is used by aiming controllers to return a TurretInput structure (TurretInput.cs), which can be passed on to any turrets attached to the vehicle for processing.

In order to define a custom AimingController, derive from the AimingController base class and implement the Initialize and Exit methods (or leave them stubbed out if not required – see examples), along with the UpdateAiming method where you return a TurretInput structure.

To use your custom controllers in the demo scene, simply place them on a GameObject under the PlayerDriverCameraRig prefab in the corresponding location. For example, place new camera controller game objects under the PlayerDriverCameraRig->Controllers->Camera object hierarchy. The name of the object will be used in the controller display UI element to indicate the current controller when selected. Name text in parenthesis will be used to show controller subtype if applicable – like “Orbit Camera (World Up)” for a world up subtype.

Turrets:

All turrets in AVK derive from the Turret base class (Turret.cs). The base class provides a single input property (Turret.input) where the turret processes aiming and firing input.

The TurretInput structure (TurretInput.cs) provides the means to communicate with a Turret. All members of the structure are nullable so that controllers can send only the information they need to. The Turret processes only the information passed to it each time. The information sent in the TurretInput structure is an aim point in world space and whether to fire the primary and secondary weapons.

In order to define a custom Turret, simply derive from the Turret base class and implement the Turret.input property setter, where you will process TurretInput parameters.

Weapons:

The base Weapons class provides some generic parameters for a weapon, including firing rate, firing accuracy (based on muzzle heat), the amount of heat added per shot and cool down times along with a heat accuracy curve, any kickback forces, and the Projectile to create along with muzzle velocity of the projectile.

To define a custom Weapon that needs extra information, simply derive from the base Weapon class and any Turrets will still be able to send firing commands as usual. See LaserBeamWeapon.cs as an example.

Projectiles:

The base Projectile class provides some generic projectile parameters including: min and max damage, min and max critical damage with critical chance, impact and explosion forces (if any), and other common parameters.

To define a custom Projectile that needs extra information, derive from the base Projectile class and any Weapons can still use the Projectile for firing.

Impact Prefabs:

Impact prefabs are just standard game objects without any special base class that the rest of the system needs to interface with. To create custom impact prefabs, simply point to your custom prefab within the Projectile's impact prefab list and specify the filter parameters as needed.

Support:

If you have any questions or issues concerning AVK, you can post your question to the AVK asset forum for help. You can also contact us at support@hebertsystems.com. Please include your **invoice number** in your email and we will get back to you as soon as we can.

Credits:

Unity Technologies:

The town models in the demo scene come from Unity's "Shanty Town" assets available on the Asset Store. Some of the textures were resized and materials consolidated between the models to reduce the size of the package.

CGTextures:

One or more textures in this package have been created with images from CGTextures.com. These images may not be redistributed by default, please visit www.cgtextures.com for more information.

Freesound.org:

Some of the audio assets in this package were created using sounds downloaded from www.freesound.org under the Creative Commons 0 license.