AGS for Unity3D

User Manual

Version 0.81 Beta

Mystik Digital 2015

Overview

Action Game System™ is an advanced model-view system for Unity3D. It is built in a modular design with many small components that can be combined in different ways depending on what type of game it is used for.

The system core is designed to fit any type of action oriented game where the player controls some kind of character. 3rd-person games, platformers and beat'em-up style games are ideal. Since it is built with a model-view design pattern, any type of camera view can be used. 3D, 2.5D and 2D games are equally supported.

AGS is currently in Beta version 0.8. Release version will include two more game demos. A 3rd-person shooter and a 2D beat'em up.

Mecanim as State Machine

With Unity5 come a wonderful thing - mecanim state machine behaviours. AGS makes heavy use of mecanim animator controllers as logic state machines (i.e. no animations involved) and uses state machine behaviours to update specific game components only when a certain state is met. This keeps the games update loop calculations at a bare minimum.

Built for coders

AGS is a system for coders, in the regard that there is no code generator tool involved. Code generating is quick for prototyping, but often slow, cumbersome and prone to errors when projects increase in size. To extend AGS natively, one must know C#.

Non-coders who want to use AGS can of course use visual scripting tools (such as PlayMaker etc.) to extend the system, or use the existing AGS components as they are. It is quite possible to create a complete game just using the AGS core and modifying the bundled AGS game demos in the Unity editor.

Reactive programming

AGS has a native lightweight library of reactive properties and lists that can be used for reactive programming. Create a public ActionProperty of any type and other classes can easily subscribe to its value change.

While useful for just about anything, ActionProperties is especially handy for making the views react independently when the data on its corresponding model are changing.

Reactive properties are also a powerful tool for storing state machine state values. With a reactive state machine, simply subscribe to the state and get notified whenever it changes.

Highly configurable

AGS does not force you into any type of corner. AGS components are of course dependent on the core, but AGS Views (the MonoBehaviours of AGS) can be mixed up with anything in the scene view, or even added to GameObjects that are already dependent on other 3rd party assets.

AGS works with *any* other asset for Unity3D. Although it may not make a perfect match with for example a car racing asset, some AGS components could be useful even for that.

Base components

GameManager

The AGS GameManager is the heart of all AGS scenes. This script is a MonoBehaviour that is placed on a root GameObject in the scene, and all other AGS Views must be attached to children of this GameObject to get properly initialized and handled by the GameManager.

The GameManager is responsible for initializing all views, both static and dynamically added views, setting up the scenes initial data, handling game state and making sure that a player is present.

Reactive Properties

Due to the reactive nature of AGS, nearly all components and systems are heavily dependent on delegates for subscriptions and notifications. AGS Views never need the regular Update method of MonoBehaviour just to overview a property. Instead AGS views react on value change notifications for the properties it is subscribing to. To understand AGS fully one must also understand delegates.

In fact, AGS views rarely use the regular Update method *at all*. Updates are mainly handled in state machine behaviours, or with the temporary update components described in a later chapter.

ActionProperty

ActionProperties are properties that can be subscribed to by any number of subscribers. AGS have two different subscribable properties. The ActionProperty<T> and the ActionList<T>. Both are completely independent components and can therefore be used in any system, but AGS use them extensively so they need to be properly understood before digging into the AGS systems.

The ActionProperty uses System.EventHandler<TEventArgs> Delegate for notification purposes and can can subscribed to by adding a delegate to its OnValueChanged event. The ActionProperty always notify new subscribers of its current value.

Note! All examples in this manual can be found in the examples folder after importing the AGS package.

Example:

Write to the debug log from a MonoBehaviour whenever an ActionProperty<int> on the model changes value.

```
PUBLIC CLASS ACTIONPROPERTYEXAMPLEMODEL
{
  PUBLIC ACTIONPROPERTY<INT> MYSUBSCRIBABLEINT { GET; PRIVATE SET; }
  PRIVATE STATIC TIMER _TIMER;

  PUBLIC ACTIONPROPERTYEXAMPLEMODEL()
  {
    MYSUBSCRIBABLEINT = NEW ACTIONPROPERTY<INT>();

    // SET UP A TIME INTERVAL FOR CALLING ONTIMEDEVENT EVERY SECOND
    _TIMER = NEW TIMER(1000) {AUTORESET = TRUE};
    _TIMER.ELAPSED += ONTIMEDEVENT;
  }
  PUBLIC VOID STARTTIMER()
  {
    _TIMER.ENABLED = TRUE;
  }
  PUBLIC VOID STOPTIMER()
  {

    _TIMER.ENABLED = FALSE;
  }
  PRIVATE VOID ONTIMEDEVENT(OBJECT SOURCE, ELAPSEDEVENTARGS E)
  {
    // INCREMENT MYSUBSCRIBABLEINT
    MYSUBSCRIBABLEINT.VALUE++;
  }
}
PUBLIC CLASS ACTIONPROPERTYEXAMPLEVIEW : MONOBEHAVIOUR
{
  PUBLIC ACTIONPROPERTYEXAMPLEMODEL MYMODELINSTANCE;

  VOID AWAKE()
  {
    MyModelInstance = new ActionPropertyExampleModel();
    MyModelInstance.MySubscribableInt.OnValueChanged += (sender, intVal) =>
    {
      // Write changed value to the debug log
      Debug.Log(intVal.Value);
    };
  }
  void OnEnable()
  {
    MyModelInstance.StartTimer();
  }
  void OnDisable()
  {
    MyModelInstance.StopTimer();
  }
}
```

For more information, see the MSDN documentation on the System.EventHandler. https://msdn.microsoft.com/en-us/library/db0etb8x(v=vs.110).aspx

ActionList

The ActionList<T> uses System.Action<T> Delegate for notification when items are added, removed or when the list is cleared. The three Actions that can be subscribed to are Action<T> ListItemAdded, Action<T> ListItemRemoved and Action ListCleared that does not have a parameter. To get notified when the list changes add a delegate to the corresponding action.

Example:

Write to the debug log from a MonoBehaviour whenever an ActionList<int> on the model get items added.

```
public class ActionListExampleModel
{
  public ActionList<int> MySubscribableList;
  private static Timer _timer;

  public ActionListExampleModel()
  {
    MySubscribableList = new ActionList<int>();
```

```
        // Set up a time interval for calling OnTimedEvent every second
        _timer = new Timer(1000) { AutoReset = true };
        _timer.Elapsed += OnTimedEvent;
    }
    public void StartTimer()
    {
        _timer.Start();
    }
    public void StopTimer()
    {
        _timer.Stop();
    }

    private void OnTimedEvent(Object source, ElapsedEventArgs e)
    {
        // Add DateTime.Now.Second to MySubscribableList
        MySubscribableList.Add(DateTime.Now.Second);
    }
}
public class ActionListExampleView : MonoBehaviour
{
    public ActionListExampleModel MyModelInstance = new ActionListExampleModel();

    void Awake()
    {
        MyModelInstance.MySubscribableList.ListItemAdded += (listItem) =>
        {
            // Write new listItem to the debug log
            Debug.Log(listItem);
        };
    }
    void OnEnable()
    {
        MyModelInstance.StartTimer();
    }
    void OnDisable()
    {
        MyModelInstance.StopTimer();
    }
}
```
For more information, see the MSDN documentation on C# Action delegates.
https://msdn.microsoft.com/en-us/library/018hxwa8(v=vs.110).aspx

Interfaces

AGS uses two Interfaces for identifying certain aspects of entities in the game.

- *IDamageable* should be implemented by any entity in the game that could be destroyed by resource damage.

- *IMovable* should be implemented by any entity that could be moved by an outside force.

Systems

If the GameManager is the heart of AGS, then the systems are its brains. All systems consist of at least one model that inherits from ActionModel and one view that inherits from ActionView. ActionModels are C# classes that doesn't rely on Unity in any way (except for their indirect use of the time based components explained on page 35). ActionViews on the other hand are MonoBehaviours, which means that they must be handled directly by the UnityEngine to work properly.

More complex systems can have several model/view pairs and also state machines that are handled by Unity 5's StateMachineBehaviours.

Available AGS systems:

- AimingSystem
- AISystem
- BaseSystem
- CameraSystem
- CharacterControlSystem
- CharacterSystem
- CheckpointSystem
- CombatSkillSystem
- GameLevelSystem
- GUISystem
- HazardSystem
- InteractionSystem
    - Interactables
    - InteractionSkills
        - LadderClimbing
        - LedgeClimbing
        - ObjectMovement
        - Swinging

        - SwicthInteraction
- MissionSystem
- MovementSystem
    - HorizontalMovement
    - Sliding
    - Swimming
    - VerticalMovement
- MovingEnvironmentSystem
- PickUpSystem
- RagdollSystem
- SkillSystem
- StatusEffectSystem
- WeaponSystem

Systems details

This section describes each system with its models and possible state machine states with more detail. Views are not included here since they are really just presenters of the data in the models.

AimingSystem

For aiming functionality when using projectile weapons.

- Models
    - Aiming
        - Aiming can be used for shooter like manual aiming (and preferably free moving/strafing) and for RPG like locked on target aiming (preferably circling around active target).
- State machine states
    - Idle
        - When player is not aiming.
    - Aiming
        - When player is aiming manually.
    - LockedOnTarget
        - When player is locked on target.

AISystem

To be used for driving enemy and NPC movement/combat/interaction skills.

- Models
    - AI
        - Inherits from CharacterControllerBase which means AI can be used by any instance of CombatEnity model. In addition to movement and action, AI can also patrol EnvironmentPaths. AI uses a state machine for its different states.
    - DetectionVolumeBase
        - Base model for detection volumes. A detection volume determines if it can detect a player on its own.
    - HearingDetection
        - Simulates hearing skill
    - SightDetection
        - Simulates vision skill
- State machine states
    - Attacking
        - When player is detected and in weapon range.
    - Chasing
        - When player is detected but not in weapon range.
    - Idle
        - If not patrolling or tracking player.
    - Patrolling
        - Only usable if AI has a set patrol route.
    - Puzzled
        - To be used when AI lost track of player or similar.

BaseSystem

Base AGS models and views.

- ActionModel (see section on ActionProperties)

- ActionView (see section on ActionProperties)

CameraSystem

A small system that provides a camera target for the main camera to follow. Camera target can switch between following the player or the player's ragdoll (if using the Ragdoll system).

- Models
  - CameraTarget
    - CameraTarget can either follow the player or the players ragdoll.

CharacterControlSystem

System for controlling a CombatEnity or anything that derives from CombatEntity (enemies, player etc).

- Models
  - CharacterControllerBase
    - Base controller that provides a MoveVector, a direction and several input buttons.
  - FloatSwitch
    - Input switch that can be used to determine input based on a float value, such as mouse wheel scrolling.
  - InputAxis
    - For x and y axis input. Can be used for a joystick.
  - InputButton
    - Input buttons can be used with Unitys GetButton, GetButtonDown and GetButtonUp, as well as custom button triggers DoubleTap and DoubleTapAndHold.

CharacterSystem

System for all types of killable, movable entities. Models inherits from each other linearly to create more complex characters.

- Models

- AdvancedCharacterBase
  - Advanced characters are characters with interaction skills.
- CharacterBase
  - Characters are essentially movable combat entities. They have movement skills plus an extra combat type skill not available to CombatEntity – the throwing skill.
- CombatEntity
  - Combat entities are killable entities that can also fight back with weapons and combat moves. Inherit from this class to create for example a defense tower.
- DamageableResource
  - The source of “energy” for Killables. There are three types of resources. Health, Stamina and Air (as in lung capacity). A resource can be vital or not. Extend resource type if other resources are needed.
- Enemy
  - This class is a straight implementation of CharacterBase.
- KillableBase
  - Base for all types of characters. Killables can not be controlled, but they have damageable resources and can be destroyed.
- Player
  - Player model inherits from AdvancedCharacter.

CheckPointSystem

System for tracking the players progress in a GameLevel.

- Models
  - Checkpoint
    - Checkpoints are meant to be used by GameLevel system. They do not store any info about the player on their own, but rather just notifies the GameLevel of when the player passes.

CombatSkillSystem

There are three types of systems for combat skills. Combat moves with equipped weapons, throwing skill and non-weapon based abilities. *Note*: Beta 0.8 does not yet have implementation for combat abilities.

- Models
  - CombatAbility
    - For non-weapon based combat skills. Not yet implemented.
  - CombatMove
    - A single move with an equipped weapon. Combat moves define a required position that CombatEntities must be in for the move to be valid.
  - CombatMoveCombo
    - A sequence of combat moves used by CombatMoveSet for combo chaining reference.
  - CombatMoveSet
    - Combat move sets define a set of combat moves and combo chains, and tracks the owner's active states and position for activating moves and chaining combos.
  - CombatSkillBase
    - Base model for all types of combat skills. Provides combat skill state, a status effect combo and several variables for defining the combat skill.
  - ThrowingSkill
    - A special combat skill that launches a throwing weapon. As opposed to other combat skills that can be used by any CombatEntity, ThrowingSkill can only be used by characters.
- State machine states
  - Charging
    - Optional state. For delaying the firing state (for example to match a combat animation) or for use with a skill that can be manually charged.
  - Firing

        - The state where the weapons appropriate HitVolume is active and tracking hits with valid targets.
    - Idle
        - Inactive state
    - Rechargning
        - All combat skills must enter a recharging state (although it can have a time window of zero).
    - SustainedFiring
        - For skills that can “auto fire”. Only transitions to recharge if out of required resources or if the fire is released.

GameLevelSystem

The game level system is an important part of an AGS scene. Although a GameLevel is not *required* to run AGS, the scene will be in some sort of limbo without a general owner. The only purpose of running AGS without a GameLevel would be to have the GameManager initialize individual views for some specific functionality, while handing over the general level mechanics to some other system.

- Models
    - GameLevel
        - Tracks player activity, handles level states and tracks objects in the level like PickUps, CheckPoints and Enemies.
    - GameLevelBound
        - Used by the GameLevel. Sets a restriction for going past a certain point in a certain direction.

GUISystem

GUI system provides functionality for display runtime GUI.

- Models
    - DynamicScreenTextBase
        - Base model for any GUI text
    - FloatingText

        - For “loose” or floating texts that use screen space world.
    - OverlayText
        - For any other type of text that is not positioned in world space but rather as screen space overlay or screen space camera.

HazardSystem

The hazard system provides death traps and environment hazards (single hit or areas). AGS also treat any type of fluid as an advanced type of hazard.

- Models
    - AreaHazard
        - Environment hazard that uses an area of effect volume for tracking its victims. Differ from single hit hazards in that they leave it over to the AreaOfEffect to track its victims.
    - DeathTrap
        - Kills killables instantly when triggered.
    - EnvironmentHazard
        - Rather than just a killing blow, environment hazards can deliver any type and number of status effects to its victims.
    - Fluid
        - Fluids are advanced area hazards that on top of any status effects it might have, it will provide buoyancy and Air damage to applicable victims.
    - HazardBase
        - Base hazard. Mainly provides the hazard state machine.
- State machine states
    - Active
        - Hazard is ready to hit applicable targets.
    - Inactive
        - Hazard has been turned off.
    - Recharging

        - Hazards can be set to recharge after being triggered, or only being active within a time interval. While recharging, the hazard is not a threat.
    - AreaHazardActive
        - Since AreaHazardViews and FluidViews do not inherit from EnvironmentHazardView, area hazards have their own active state. FluidViews inherit from AreaHazardView and should also use the AreaHazard states.
    - AreaHazardInactive
        - See above.
    - AreaHazardRecharging
        - See above.

InteractionSystem

Interaction system is divided into two main categories. Interactables and interaction skills. Each interaction skill can interact with a specific interactable. All interaction skills use share interaction volumes for detecting interactable targets.

- Models
    - InteractionSkillBase
        - Base model for interaction skills
    - InteractionSkills
        - A bundle of all available interaction skills plus any number of interaction volumes.
    - InteractionVolume
        - Interaction volumes are used to detect interactables.
- Interactable Models
    - InteractableBase
        - Base model for any interactable.
    - Ladder
        - Just a reference model for ladder interaction.
    - Ledge
        - Just a reference model for ledge interaction.
    - MovableObject

        - Movable objects implement the IMovable interface and thus can be affected by push effects. They have different states depending on character interaction.
            - State machine states
                - Carried
                - Grabbed
                - Idle
                - PickedUp
                - Thrown
    - Swing
        - The swing consists of a chain of SwingUnits. A swing is only indirectly controlled by a swinging character.
    - SwingUnit
        - One unit in a Swing. To be gripped by a swinging character. SwingUnits have different states depending on character interaction.
            - State machine states
                - Idle
                - ReducingSpeed
                - SwingingNatural
    - Switch
        - Switches can be on or off.
    - UnlockableObject
        - An UnlockableObject consists of any number of Switches and unlocks if all switches are switched on. Unlockable objects are not locked or unlocked directly by a switch interacting character.
- InteractionSkills models
    - LadderClimbing
        - For climbing interactable ladders.
        - State machine states
            - Approaching
                - Initial state when interacting, use this state to place the character in the correct position in regard to the ladder.
            - Climbing

        - ♦ Climb the ladder up or down.
      - ExitingTop
        - ♦ Use this state if need to position the character after leaving the ladder on top.
      - Idle
        - ♦ When not interacting with a ladder.
      - Jump
        - ♦ Jumping off the ladder
      - Releasing
        - ♦ Releasing the grip on the ladder.
- ➢ LedgeClimbing
  - ▪ For hanging, climbing and jumping off ledges.
  - ▪ State machine states
    - Approaching
      - ♦ Initial state when interacting, use this state to place the character in the correct position in regard to the ledge.
    - Climbing
      - ♦ For climbing up when hanging on a ledge.
    - Grabbing
      - ♦ This is a sort of idle state when hanging from the ledge.
    - Idle
      - ♦ When not interacting with a ledge.
    - Jumping
      - ♦ For jumping backwards when hanging on a ledge.
    - Releasing
      - ♦ For dropping down when hanging on a ledge.
- ➢ ObjectMovement
  - ▪ For grabbing, pushing, pulling, carrying and throwing MovableObjects. Create different behaviour based on current movable objects weights or sizes. For example - let the character only be allowed to pick up and throw tiny objects, or prohibit pull of heavy objects so that they can only be pushed forward.
  - ▪ State machine states

        - Approaching
            - ♦ Initial state when interacting, use this state to place the character in the correct position in regard to the movable object.
        - Carrying
            - ♦ Lift up movables and carry them.
        - Grabbing
            - ♦ Grab the movable and start to push or pull it.
        - Idle
            - ♦ When not interacting with a movable object.
        - Releasing
            - ♦ Release the grip on the movable object.
        - Throwing
            - ♦ Throw the movable object when its being carried.
- ➢ Swinging
    - ▪ For grabbing a SwingUnit and start swinging or climbing the Swing.
    - ▪ State machine states
        - Approaching
            - ♦ Initial state when interacting, use this state to place the character in the correct position in regard to the swing unit.
        - Climbing
            - ♦ Climb up or down the swing, switching current gripped SwingUnit
        - Idle
            - ♦ When not swinging.
        - Jumping
            - ♦ Jump off the swing.
        - Releasing
            - ♦ Release and drop down from the swing.
        - StopSwinging
            - ♦ For stopping the swing from swinging.
        - Swinging

            - ♦ A sort of idle state when grabbing a swing unit and swinging.
- ➢ SwitchInteraction
    - ▪ For turning switches on or off.
    - ▪ State machine states
        - Approaching
            - ♦ Initial state when interacting, use this state to place the character in the correct position in regard to the switch.
        - Idle
            - ♦ When not interacting with a switch.
        - Switching
            - ♦ When approaching state is done, trigger the interact with the switch.

MissionSystem

MissionSystem consist of missions and mission objectives.

- ❖ Models
    - ➢ Mission
        - ▪ Missions are based on its mission objectives. GameLevel can use missions for game level states depending on mission state.
    - ➢ MissionObjective
        - ▪ A mission objective can be required or optional, and completed or failed.

MovementSystem

This system provides the core skills of a character. Provides horizontal and vertical movement, as well as sliding and swimming.

- ❖ Models
    - ➢ MovementSkillBase
        - ▪ Base model for movement skills

- MovementSkills
  - A bundle of all available movement skills
- MovementSkills models
  - HorizontalMovement
    - For moving the character in different modes in any horizontal direction.
    - State machine states
      - Crouching
        - For crouched movement.
      - Idle
        - When standing still
      - Moving
        - For normal upright movement.
      - Sneaking
        - For sneaky movement.
      - Sprinting
        - For extra fast movement.
  - Sliding
    - For helping the character walk on, stand still or slide down slopes.
    - State machine states
      - HelplessSliding
        - For steep slopes and disabling other skills.
      - Idle
        - When not on a slope.
      - ManualSliding
        - For a slide based on input.
      - NaturalSliding
        - Let physics work normally.
      - PreventSliding
        - Useful for RigidBody characters that normally slide backwards even on slightly angled slopes.
  - Swimming
    - For movement in Fluids.

        - State machine states
            - DoingStroke
                - Do a swimming stroke.
            - InFluid
                - When within a fluids area of effect.
            - OutOfFluid
                - When not in a fluid.
            - SurfaceJumping
                - For jumping out of the fluid when on the surface.
    - VerticalMovement
        - For any time of vertical movement. Jumping, falling, wall jumping etc. Supports combo jumping.
        - State machine states
            - Falling
                - When not grounded and not jumping.
            - Idle
                - After landed when grounded.
            - Jumping
                - For manually jumping up.
            - Landing
                - When just landing on ground. Used for determining combo jumping timer.
            - WallJumping
                - For jumping off walls while airborne.

MovingEnvironmentSystem

This system consists of two parts. The environment path (and points) and the moving environment. Any moving environment can use any environment path. Also – AI's can use environment paths as a patrol route. There is also a utility widget script available (DrawEnvironmentPathPoints) that plots any environment path in the Unity editor.

- Models

- EnvironmentPath
  - The path is made of environment points. They can be set to either circulate or ping pong its moving environments and/or AI's.
- EnvironmentPoint
  - A single point on a path.
- MovingEnvironment
  - As opposed to AI's, moving environments require an environment path. They can navigate the path forward or backward, and move with linear, lerped or slerped movement between points.
  - State machine states
    - Move
      - When the moving environment is navigating the path.

PickUpSystem
This system provides different pick ups for the player.

- Models
  - PickUpItemBase
    - Base pick up item.
  - PickUpStatusEffect
    - PickUps that hands out any number and type of status effects for the player.
  - PickUpThrowable
    - Adds throwable weapons of given type and count to the player.
  - PickUpWeapon
    - Adds a new weapon to the player. Can also provide a CombatMoveSet to go with it.

RagdollSystem
This system is tightly coupled with Unitys Ragdoll. It uses the RagdollHelperScript to instaniate ragdolls that are exactly synced

both with position and velocity where their “living” counterparts died.

- Models
  - Ragdoll
    - Corresponds to a Unity Ragdoll, with methods for syncing the ragdoll. Ragdolls also implement the IMovable interface and can therefore be pushed by push effects.
  - PlayerRagdoll
    - Just like any ragdoll, but a child model of Ragdoll so that it can be easily found and separated from other ragdolls (usable for the camera target).

SkillSystem

This system provides a base skill model for *any* skill in AGS. Movement skills, combat skills, interaction skills, all inherit from base skill.

- Models
  - SkillBase
    - This model provides functionality for enabling/disabling the skill and takes care of resource management for skills that require resources to use.

StatusEffectSystem

AGS status effects can be divided into two main groups of effects, plus a third Unity special effect that is the explosion effect. The two groups are periodic effects and static effects. Peridiodic effects can have ticking intervals, while static effects are applied with a given timespan (or indefinitely) and are then constantly active until they run out or are removed. Any object that use status effect(s) can also use the AreaOfEffect model for applying the effect(s) to multiples targets.

- Models

- AreaOfEffect
  - While not exactly a status effect on its own, this model provides multiple targets functionality that can be used in combination with status effects. It tracks applicable targets within its volume for the owner model to apply its effects to.
- ContinuousResourceEffect
  - A special ticking resource effect. To be used with sustained skills that constantly drains a resource effect while active, or similar.
- ExplosionEffect
  - A Unity special effect. This is the only effect that does not inherit from StausEffectBase. Its variables directly correspond to Unitys function AddExplosionForce.
- MovementEffect
  - A static status effect. MovementEffect is primary used to affect a characters movability.
- PeriodicStatusEffectBase
  - Base model for periodic effects. Provides tick count and the time interval between ticks.
- PushEffect
  - The PushEffect is periodic effect used for pushing IMovables around the game level.
- ResourceEffectCombo
  - Just a bundle of ResourceEffects and ContinuousResourceEffects.
- StaticStatusEffectBase
  - Base model for effects that should be applied as constant effects for its duration.
- StatusEffectBase
  - Base model for all status effects except ExplosionForce. Provides a strength modifier and the type of strength to be applied. Strength can be fixed or percentage based.
- StatusEffectCombo
  - A bundle of any type and number of StatusEffects. A skill, weapon or hazard typically owns a StatusEffectCombo.

- SuperNaturalEffect
    - These static type special effects that can be applied to Killables. Existing super natural effect types are invulnerability and damage resistance. Could potentially be used for anything that does not involve moving or pushing a target.

WeaponSystem

The weapon system covers equipable weapons as well as throwable weapons and projectiles.

- Models
    - EquipableWeaponBase
        - Equipables weapons are held by a CombatEnity (unarmed is also considered a weapon) and swung/activated/fired with the help of combat moves. These weapons can have multiple hit volumes where one at a time are activated based on combat move states and hit volume indexes.
    - HitVolume
        - Defines a hit volume for a weapon. Tracks targets for the weapon to apply its effects to during firing state.
    - MeleeWeapon
        - The most basic equipable weapon. Use combat moves to swing it.
    - Projectile
        - Fired by a projectile weapon, a projectile can hit targets with a StatusEffectCombo (and/or an ExplosionEffect) on its own. With projectiles it is possible to hit a target with three different status effect combos at the same time. One from the projectile, another from the projectile weapon that fired the projectile, and yet another from the combat move that was used to fire the weapon. For example – one could do a head shot move for extra damage output by lowering the targets defences, with a sniper rifle that leaves the target bleeding and limping, and on top of that use exploding

bullets that also sends them flying. Just about anything is possible.

- ProjectileWeapon
  - These weapon can be swung like melee weapons, but are also capable of firing projectiles.
- ThrowableWeapon
  - This special weapon is thrown instead of held or fired. Thrown weapon can be set to trigger after a certain time and/or when hitting a target. They can even be set to stick to the targets they hit.
- WeaponBase
  - Base model for all weapons. Provides a StatusEffectCombo which makes it possible to hit a target with two simultaneous effects combos. One from the weapon itself and another from the combat move that was used to activate the weapon. *Note*: this does not apply for throwable weapons, since the throwing skill is only used for launching the throwable weapon and does not have a status effect combo.

Creating a new system

To extend AGS with a new system, the very least needed is a model that inherits from ActionModel and a view that inherits from ActionView. The model holds that data for the task to be solved. The view is responsible for creating an instance of the model. This is done in the InitializeView() method, which is run automatically but the GameManager since the view is a child of ActionView.

After creating the model instance, the view should also call SolveModelDependencies with its model instance as parameter to solve any dependencies to other models/views. If the model itself is dependent on other models, the view should override the SolveModelDependecies method to set the dependency references. Finally, to safely subscribe or operate on the models data, the view can override InitializeActionModel method. This is also called by the

GameManager after InitializeActionView, and at this point all dependencies are solved.

The following example shows a small but useful system that provides pressure plate type functionality for the player.

Example:

A simple system that creates a pressure plate for the player.

Model:

```
public class PressurePlate : ActionModel
{
  // Subscribable properties
  public ActionProperty<bool> IsPressured { get; set; }

  public PressurePlate()
  {
    IsPressured = new ActionProperty<bool>();
  }

  public void TriggerMechanism(bool on)
  {
    IsPressured.Value = on;
  }
}
```

View:

```
public class PressurePlateView : ActionView
{
  public PressurePlate PressurePlate;

  public override void InitializeView()
  {
    PressurePlate = new PressurePlate();
    SolveModelDependencies(PressurePlate);
  }

  public override void InitializeActionModel(ActionModel model)
  {
    base.InitializeActionModel(model);

    // Set up an action delegate for on trigger enter action with a player
Action<PlayerBaseView> pressureOnAction = playerView =>
PressurePlate.TriggerMechanism(true);
    gameObject.OnTriggerActionEnterWith(pressureOnAction);
```

```
      // SET UP ANOTHER ACTION DELEGATE FOR ON TRIGGER EXIT ACTION WITH A PLAYER
ACTION<PLAYERBASEVIEW> PRESSUREOFFACTION = PLAYERVIEW =>
PRESSUREPLATE.TRIGGERMECHANISM(FALSE);
      GAMEOBJECT.ONTRIGGERACTIONEXITWITH(PRESSUREOFFACTION);
   }
}
```

ViewScripts

ViewScripts are small MonoBehaviours that is meant to be used as a supplement to ActionViews, to provide modularity, reusability, testability, readability and reduce file length.

ViewScripts own a reference to an ActionView, and typically subscribe to a property on this views model and then operate on its own to provide extra functionality. A useful example of a ViewScript is to separate FX or animation logic from the ActionView.

In some cases it can also be useful for the ActionView to know of a ViewScript. For example, if the ViewScript is responsible for creating a particle system, the ActionView may want to know when the particle system is finished.

AGS Core contains dozens of ViewScripts, each script with a single specific purpose. Here is a list and a short description of all ViewScripts:

Misc ViewScripts

- *ContinuousResourceRegen*: Add this to a Killable entity to add a regen to a resource of choice.
- *PlayerNoiseVolume*: Script that simulates player sound with a sphere collider. Collider is disabled when player is silent.
- *SlowWhenDamaged*: Slows a character down when health drops to a specific threshold.

Targeting ViewScripts

- *TargetingSystem*: Keeps track of nearby enemies.
- *WeaponAim*: For aiming projectile weapons towards target.

FX ViewScripts

- *CombatMoveSetFX*: Handles particles and/or sound for a specific combat moveset.
- *EnemyFX*: Handles footstep sound for enemies.
- *GameLevelFX*: Handles game level sound effects.
- *GameLevelMusic*: Handles game level music.
- *HazardFX*: Handles particles and/or sound to Hazards on circumstances of choice.
- *PlayerFX*: Handles player sounds and particle systems.
- *ProjectileWeaponFX*: Handles a projectiles sound and particle systems.
- *SwitchFX*: Handles a switch’s sound
- *ThrowableWeaponFX*: Handles a throwable weapons sound and particle systems.
- *UnlockableObjectFX*: Handles an unlockable objects sound.

Animation ViewScripts

- *CombatSkillAnimations*: Drives a mecanim animator for character animation based on combat move states and character input.
- *InteractionSkillAnimations*: Drives a mecanim animator for advanced character animation based on interaction skill states.
- *MovementSkillAnimations*: Drives a mecanim animator for character animation based on movement skill states.
- *SwitchAnimations*: Drives a mecanim animator for a switch’s animation.
- *ThrowingSkillAnimations*: Drives a mecanim animator for character animation based on throwing skill states.
- *UnlockableObjectAnimations*: Drives a mecanim animator for an unlockable object’s animation.
- *WeaponAnimations*: Drives a mecanim animator for weapon animation based on combat move states and character input.
- Anim event listeners (For use with mecanim animation events)
  - *ProjectileWeaponAnimEventListener*: References a projectile weapon. Listens for a Fire() event and fires referenced projectile weapon.
  - *ProjectileWeaponMoveAnimEventListener*: References a combat entity. Listens for a Fire() event and fires the combat entity equipped projectile weapon.
  - *ThrowingSkillAnimEventListener*: Listens for the ThrowArc() or ThrowForward events and begins the appropriate throwing skill.

Example:

A ViewScript that references a PressurePlate and changes the material color of the pressure plate when it's IsPressured property changes.

```
public class PressurePlateColor : ViewScriptBase
{
  private PressurePlateView _pressurePlateView;
  private PressurePlate _pressurePlate;
  private Renderer _renderer;
  public override void Awake()
  {
    base.Awake();
    _renderer = GetComponent<Renderer>();
  }

  protected override void SetupModelBindings()
  {
    base.SetupModelBindings();
    if (ViewReference != null)
    {
      _pressurePlateView = ViewReference as PressurePlateView;
      if (_pressurePlateView != null)
      {
        _pressurePlate = _pressurePlateView.PressurePlate;
      }
    }
    if (_pressurePlate == null) return;

    _pressurePlate.IsPressured.OnValueChanged += (sender, isPressured) =>
    {
      if (_renderer != null)
      {
        _renderer.material.SetColor("_Color", isPressured.Value ? Color.green : Color.red);
      }
    };
  }
}
```

Miscellaneous components

MonoExtensions

AGS has three sets of classes that extends MonoBehaviour with extra functionality for component handling, collision handling and trigger handling.

ComponentExtensions is a class that extends MonoBehaviour with some static methods for component functionality.

Methods:

- bool *HasComponent*<T>(GameObject gameObject**)**
  - Returns true if gameObject has a MonoBehaviour of type T.

- T *SetupComponent*<T>(GameObject gameObject)
  - Looks for a MonoBehaviour of type T on gameObject. Adds Component T if not already present. Returns T.

- T *AddComponent*<T>(GameObject gameObject)
  - Adds Component T to gameObject. Returns T.

- T *AddComponentOnEmptyChild*<T>(GameObject gameObject, string childName = “”)
  - Adds Component T on a new child GameObject to gameObject with optional childName. Returns T.

- void *RemoveComponent*<T>(GameObject gameObject, T component)
  - Removes component T from gameObject.

TriggerExtensions is a static class that can be used to automatically setup collider trigger functionality with a specific Component on a GameObject.

Methods:

- void *OnTriggerActonEnterWith*<T>(this GameObject gameObject, Action<T> action**)**

    - Sets up a trigger enter action for gameObject with another GameObject that has a Component of type T. When trigger occurs, the other GameObjects T is returned.

- void *OnTriggerActonStayWith*<T>(this GameObject gameObject, Action<T> action**)**
    - Sets up a trigger stay action for gameObject with another GameObject that has a Component of type T. When trigger occurs, the other GameObjects T is returned.

- void *OnTriggerActonExitWith*<T>(this GameObject gameObject, Action<T> action**)**
    - Sets up a trigger exit action for gameObject with another GameObject that has a Component of type T. When trigger occurs, the other GameObjects T is returned.

CollisionExtensions is a static class that can be used to automatically setup collider collision functionality with a specific Component on a GameObject

Methods:

- void *OnCollisionActonEnterWith*<T>(this GameObject gameObject, Action<T> action**)**
    - Sets up a collision enter action for gameObject with another GameObject that has a Component of type T. When collision occurs, the other GameObjects T is returned.

- void *OnCollisionActonStayWith*<T>(this GameObject gameObject, Action<T> action**)**
    - Sets up a collision stay action for gameObject with another GameObject that has a Component of type T. When collision occurs, the other GameObjects T is returned.

- void *OnCollisionActonExitWith*<T>(this GameObject gameObject, Action<T> action**)**
    - Sets up a collision exit action for gameObject with another GameObject that has a Component of type T. When collision occurs, the other GameObjects T is returned.

TimerComponents

Timer components are classes that simplifies timed activities, intervals and temporary updates. They can be easily added to GameObjects in the scene at runtime with the help of ComponentExtensions methods.

TimerComponent are MonoBehaviours that uses Coroutines to provide timer and interval functionality. There are two types of TimerComponent. The TimerPersistantGameObject component and the TimerTemporaryGameObject component. The difference is that TimerPersistantGameObject only destroys the TimerComponent when the timer is finished. TimerTemporaryGameObject destroys the GameObject. Action TimerMethod describes the callback method to run when timer is done.

Example:

Add a temporary timer to a gameObject:

```
var myTempTimer =
ComponentExtensions.AddComponentOnEmptyChild<TimerTemporaryGameObject>(gameObject,
"My Temporary Timer");
myTempTimer.TimerMethod = () => Debug.Log("Timer is finished!");
myTempTimer.Invoke(1f);
```

UpdateComponent are MonoBehaviours that should be used for temporary updates. There are two types of TimerComponent. The UpdatePersistantGameObject component and the UpdateTemporaryGameObject component. The difference is that UpdatePersistantGameObject only destroys the UpdateComponent after finished updating. UpdateTemporaryGameObject destroys the GameObject. There are also similar FixedUpdate components.

Example:

Add a temporary update component to a GameObject:

```
var myTemporaryUpdate =
ComponentExtensions.AddComponentOnEmptyChild<UpdateTemporaryGameObject>(gameObjec
t, "My temporary update");
myTemporaryUpdate.UpdateMethod = () =>
```

```
{
  Debug.Log("This is called every update until myTemporaryUpdate is stopped!");
};
```

UIComponents

UIComponents are MonoBehaviours that own a reference to the GameLevelHUDView and uses Unity GUI is some way to provide a ready made UI widget.

- TextCounters are simple scripts that reads a value and updates a UnityEngine.UI.Text.

- MissionHUD is a widget that tracks current GameLevel mission and objectives and shows them with UnityEngine.UI.Text.

- ResourceBars are widgets that uses UnityEngine.UI.Image to display a DamagableResource by filling the image with the percentage of the current resource value / resource max value. There is PlayerResourceBar where the type of resource to display can be selected, and TargetResourceBar that displays the name and health of Players current target.

Demo games

Space Ninja

Space Ninja is a 2.5D platformer demo game. The game camera is orthogonal and the player character is a controlled Rigidbody. Player and enemy deaths are handled with ragdolls.

This demo features most of the AGS components. The environment has death traps, area hazards, mines, a moving death ball and several angry robot guards. To reach the exit, the player character must run, sprint, sneak, crouch, slide, swim, jump, combo jump, wall jump and ride on moving platforms. Ninja must also use all types of interaction skills.

Space Ninja also makes use of the combat system. This involves weapon handling, combo chaining and combat move enabling/disabling depending on the current character position. Alongside unarmed combat, this demo also features one extra weapon for the player. A bazooka.

Ninja also has a throwing skill that can be used with different throwing weapons. Player starts out with grenades but can also pick up a sticky bomb type weapon along the way.

Game controls:

Movement		*Combat*	
Left	*A*	*Fast attack*	*Mouse Button 1*
Right	*D*	*Heavy attack*	*Mouse Button 2*
Climb/Grab	*W*	*Throwing weapon*	*Mouse Button 3*
Crouch/Slide	*S*	*Switch weapon*	*Mouse Wheel or Q*
Jump	*SPACE*	*Switch throwing weapon*	*T*
Interact	*E*		
Sprint	*Double tap and hold A or D*	*Other*	
Sneak	*X*	*Pause*	*ESC*

Planned demo games:

3D shooter
2D beat’em up