SmoothTurret

Script Reference

There are seven major script components to SmoothTurret. Most of which you can replace with your own implementations.
More detailed reference can be found in the Script Reference document.

ST_Turret
The main control for the turret motion. It also has functions to query if the turret is aimed at the target, or if a given target is within the turret's range of motion. This should be placed at the root level of your turret object.

MF_BasicWeapon
Contains the properties of a weapon, and has functions to determine if the weapon is ready to fire, and a command to shoot. Place this at the root level of your weapon object. This could be replaced with your own weapon script.

ST_TurretControl
This bridges ST_Turret script and the MF_BasicWeapon script. Turret Control will ask the turret if it is pointed at the target. If so, then asks the weapon if it is ready to fire. If yes, then sends the command to shoot one or more of the turrets weapons. Place this on the same object as the ST_Turret script, at the root level of your turret.

MF_BasicScanner
Will detect targets within a particular range using tags or layers to define teams, and store them in a list of potential targets, with info about those targets, in MF_TargetList. MF_BasicTargeting will then search the list for an appropriate target.

MF_BasicTargeting
Holds the logic for choosing targets from a list generated by MF_BasicScanner. This will typically be placed on the same object as ST_Turret.

MF_TargetList
Holds the targets found using MF_BasicScanner. This will typically be at the root level of the game unit. As more than one scanner may contribute to it, and more than one turret may access it, in the case of a complex unit with multiple scanners and/or turrets.

Base Classes and Static Functions
MF_StaticCompute

This contains much of the math to produce the turret motion and to compute an intercept point. You won't need to attach this to an object. Just make sure it's in your project hierarchy.

MF_StaticMath
A few math utilities. A proper modulus operator (as opposed to the remainder operator used in UnityScript) And an AngleSigned function.
Mod (***a***:int, ***n***:int) : int
Returns *a* modulo *n*, and works with negative numbers.
AngleSigned (***v1***:Vector3, ***v2***:Vector3, ***axis***:Vector3)
Returns the positive or negative angle between two direction vectors in relation to an *axis*.

MF_BasicTargetData
This is the class definition used in MF_TargetList. It is separate in the event that you want to add more complex targeting behaviors, which would need more complex information gathered and stored by the scanners.

MF_AbstractPlatform
This is the base class for ST_Turret. This could allow allow MF_BasicTargeting to be used for other purposes besides a turret, such as navigation.

MF_AbstractTargeting
This is the base class for MF_BasicTargeting. This could allow different or more advanced targeting routines to be swapped out without having to change ST_TurretControl.

Supporting Scripts
In addition, there are a few supporting scripts to round out the package. Most likely, you won't use these in your project. As least not without some modification.

MF_BasicProjectile
A simple bit of code for the projectiles fired by the weapons. It uses raycast to check ahead for colliders to determine hits to compensate for fast movement speeds. It will destroy itself after a fixed time, or if it detects a hit. The duration is supplied by the weapon script upon firing. Damage is applied to a hit target if it has the ST_Status script. It will also display a 'blast' object upon a collision.

ST_Player
Just a basic player script to track if the player is controlling eligible turrets or not. You can define an object to be an aim point, but if none found, it will create an empty object to use as one.

MF_BasicStatus
Another really basic script containing a health variable for targets. If health <= 0 then it will destroy the object.

MF_BasicNavigation

A simple script to move the targets to various waypoints.

You can also designate a group of waypoints, by arranging them as children of game object, and assigning that object in *waypointGroup*.

MF_BasicSelfDestruct

Destroys its GameObejct after a designated amount of time.

ST_SpawnEnemyDemo

Creates enemies for the turrets to shoot at. This is just for demonstration purposes.

ST_Turret : MF_AbstractPlatform

This is the main turret control script, and it will control the aiming of your turret. Place this in the root GameObject of your turret.
Typically, this will be aiming a weapon at a target, but can be used with cameras, or anytime you want to aim a part using realistic rotation.

Public Functions

Inherited from MF_AbstractPlatform

TargetWithinLimits (***target*** : Transform)
Returns true if *target* is within the turret's range of motion.

Native

AimCheck (***targetSize*** : float, ***aimTolerance*** : float)
Returns true if the turret is pointed at its current target.
The turret must aim within a circle of *targetSize* size at the location of the target.
aimTolerance will add or subtract extra degrees radius to the apparent size of the target.

Public Variables

Inherited from MF_AbstractPlatform

target : GameObject
The target the turret is currently attempting to track. This is supplied by ST_TurretControl.

weaponMount : GameObject
This is the part that the weapon is attached to. It is used in several different aiming calculations. When using multiple weapons, this should be centered between them.

Native

useIntercept : boolean
SmoothTurret can optionally calculate the linear intercept point to shoot a projectile at a target. To do so, the turret will need the shot speed of the projectile to be fired, and this projectile should move at a constant rate.
By default *shotSpeed* is supplied by the ST_TurretControl script.
Be aware that there are a number of factors that can result in a miss. First, the is the accuracy of the turret and the weapon. Second, if the target changes speed or direction during the shot time of flight, this can also result in a miss.

rotationRateMax : float (1 to 1000)
elevationRateMax : float (1 to 1000)
The maximum rate in degrees per second that each part may rotate. Note, that often times your turret may not reach higher maximum turn rates. This is due to two main factors. First, the turret will also need time to slow down at some point before it reaches its destination, effectively limiting how fast it can turn. Second, there will typically only be 180 degrees max to rotate to its goal. Between the time needed to accelerate and slow down, there may not be enough degrees to hit the max turn rate before reaching its goal.
These will also have an effect on the pitch/rate of the rotator and elevator motion sounds.

rotationAccel : float (1 to 1000)
elevationAccel : float (1 to 1000)
The acceleration rate of the rotator and elevator parts of the turret. This is how fast that part speeds up and slows down its turning. Accelerate rates above 500 will begin to cause some slight jitter, with it getting more noticeable around 700. High acceleration rates also begin to suffer diminishing returns. To get true high speed turning, consider using constant turn rate, as this will allow very fast and precise turning, as well as eliminating high speed jitter.

decelerateMult : float
This allows you to cause deceleration to be a different rate than accelerating. It affects both rotation and elevation. Typically, this value will be used to simulate braking being easier than accelerating.

limitLeft : float (0 to -180)
limitRight : float (0 to 180)
limitUp : float (0 to 90)
limitDown : float (0 to -90)
Each of these allow you to place traversal limits on the rotator and elevator parts. Movement tracking left and down use negative numbers. Tracking to the right and up use positive numbers.

restAngle : Vector2
You may define an angle for the turret to point when it has no target.
x is the angle of elevation (-90 to 90).
y is the angle of rotation (-180 to 180). 0, 0 points straight ahead.

aimError : float
aimErrorInterval : float
Optionally, you can give the turret some inaccuracy in rotation. *aimError* is the inaccuracy in degrees, and *aimErrorInterval* is how often a new random 'wrong' position is calculated. These values typically look best when under 1, with the interval looking better higher when turn rates are slower.

turningAimInaccuracy : float

Adds to turret *aimError* if the turret is turning, for every degree per second, from both the elevator and rotator parts. Be aware that the movement of the turret to the error position, will also cause error to increase, as this motion itself is considered movement. Small values work better here.

turningWeapInaccuracy : float
Adds to weapon inaccuracy if the turret is turning, for every degree per second, from both the elevator and rotator parts. ST_TurretControl will send this value to every weapon in the turret. But will be bound by each individual weapon's *minAccuracy,* and *maxInaccuracy* found in ST_TurretControl.

constantTurnRate : boolean
This allows you to forgo smooth natural movement, and have the turret simply use constant turn rates. This setting uses less computation, and is also better for turrets using very fast turn rates. Accuracy is also better than using smooth turning.

If this option is used, the turret will turn at the rate supplied in *turnAccel* and *elevatorAccel* but will still be limited by *turnRateMax* and *elevatorRateMax*.

rotator : GameObject
The part that does the actual rotation.

elevator : GameObject
The part that does the elevation.

rotatorSound : AudioSource
elevatorSound : AudioSource
These point to the AudioSource to be used for each part's movement sound. This sound will play at a varying pitch/rate based on the current rotation/elevation speed as compared to its respective maximum. At half the maximum turning rate, it will play at the normal pitch. It will go down as it approaches 0 rate, and up as it approaches max rate. The max and min pitch values are defined by *soundPitchRange*.
For best results, this sound should be a constant looping whir or hum. Steady looping rhythmic elements work good too, as their rate will also vary with the turning speed.

soundPitchRange : float
This is how much is added or subtracted from the pitch/rate of the rotatorSound and elevatorSound.

platform : GameObject
Optionally, you may supply the object that holds the turret, or the root object of the character or entity. This will be used to find the velocity of that object to be used in intercept calculations. If not using intercept, or if that object is stationary, you can leave this blank, and SmoothTurret will assume velocity is 0. If the supplied object

has a rigidbody, it will use the rigidbody velocity. If no rigidbody is found, velocity will be computed from the object’s change in position.

Hidden Variables

Inherited from MF_AbstractPlatform

shotSpeed : float
This is used to calculate an intercept path for fired shots. This can be set manually if this value isn’t going to change, but this will typically be supplied by another script. SmoothTurret comes with the ST_TurretControl script that will automatically supply this value from the weapon.

targetLocation : Vector3
The location of where the turret is to aim, not the actual location of the target. This can be different due to *useIntercept* calculations or *aimError,* and *turningAimInaccuracy*.

Native

targetRange : float
Distance to the target.

platformVelocity : Vector3
The velocity of the platform object. If *useIntercept* is false, then this value will be incorrect and ignored.

systAimError : Vector3
The current location of any error introduced with aimError. This will change every *aimErrorInterval* seconds.

averageRotRateEst : float
An estimated rotation rate per second averaged from this frame and the last.

averageEleRateEst : float
An estimated elevation rate per second averaged from this frame and the last.

totalTurnWeapInaccuracy : float
The amount of inaccuracy ST_FireControl will send to weapons resulting from *turningWeapInaccuracy* * (*averageRotRateEst* + *averageEleRateEst*)

totalTurnAimInaccuracy : float
The amount to add to *aimError* resulting from *turningAimInaccuracy* * (*averageRotRateEst* + *averageEleRateEst*)

ST_TurretControl

SmoothTurret is set up to be modular, so that if you would like to replace the weapons or write your own control routine, you can do so without having to re-write the turning mechanism or other parts.

ST_TurretControl will allow for AI or Player control of the turret. It bridges communication between the turret, weapon, and targeting scripts, and is placed on the same object that holds the ST_Turret script. This is designed to be used with the MF_BasicWeapon, and MF_BasicTargeting scripts, so if you wish to use another weapon script, a new fire control script will be needed to interface it with the turret.

In addition, it will also read any inaccuracy caused from turret rotation and elevation, and send it to the weapons to simulate degradation of aim while parts are in motion.

In short, this script holds pointers to the weapons this turret has, and includes some aiming and behavior information. It asks the turret if it is aimed at the target. If it is, it then asks the weapon if it is ready to fire. If so, then sends a fire command to the weapon.

Public Variables

target : GameObject
This is the object sent to the turret as it's target. Typically this will be gathered from MF_BasicTargeting when under AI control, but can be also used with another script.

controller : ControlType enum { AI_AutoTarget, AI_NoTarget, Player }
AI_AutoTarget: Will use a target from MF_BasicTargeting and fire when the turret is aimed correctly.
AI_NoTarget: Will not gather a target. But if a target is supplied, the turret will aim and fire at it.
Player: Turret will aim at the defined player's aimObject, and will fire when the player presses the left mouse button. That player must also have *turretControl* set to true. If false, the turret will not fire or move.

player : GameObject
Used when *controller* is set to Player. This will give the turret that player's *aimObject* as its target.

weapons : WeaponData[] class { ***object*** : GameObject, hidden(***script*** : ST_Weapon) }
This array holds the root GameObject for each weapon the turret holds. For best results, each weapon should be the same, but this is not necessary. Select the number of weapons this turret has, and then drag each weapon into a field.

fixedConvergeRange : float

When using multiple weapons, you may designate a range that weapon fire will converge at using *fixedConvergeRange*. This will make the weapons angle slightly during Start() at runtime.

dynamicConverge : boolean
If true, weapons will ignore *fixedConvergeRange*, and continuously angle to produce convergence at the range of the current target.

convergeSlewRate : float
Determines how fast the individual weapons will angle in degrees per second during *dynamicConverge*.

minConvergeRange : float
You may also designate a minimum convergence range, so as to limit the amount that the individual weapons can move.

alternatingFire : boolean
When using multiple weapons, you can choose to have them all fire at once, or alternate shots between them. If alternating, the delay before the next weapons fires is the *cycleTime* of the weapon at index 0 divided by the number of weapons. This will produce odd results if you have weapons with different rates of fire.

checkLOS : boolean
losCheckInterval : float
Select if the turret should use line of sight to determine whether or not to fire. If performance becomes an issue, you can define an interval to wait between checks, although this can result in the weapon being less responsive to rapidly changing line of sight conditions.

MaxInaccuracy : float
If the turret’s *turningWeapInaccuracy* is > 0, inaccuracy will be sent to the weapons. This will limit the amount of inaccuracy a weapon can have.

checkTargetSize : boolean
This script tries to determine the target size when choosing whether or not to fire. This way, it will fire even if not aimed directly at the target center. It approximates the size of the target using its largest dimension of any root collider component. If no collider is found, it will look for a collider in the root child index 0. If still none found, it will use *targetSizeDefault*.

targetSizeDefault : float
If no target size can be found, you can set a default size for the the weapons to fire at.

Hidden Variables

curWeapon : int

The index number of *weapons* that is the next weapon to fire.

MF_BasicWeapon

Place this at the root level of a weapon GameObject. This controls all the properties of a weapon. It is currently accessed via the ST_TurretControl script. If you wish to use a different weapon script, you will need to change or replace ST_TurretControl.

Public Functions

Shoot ()
Call this function when the weapon should shoot. It will internally call ReadyCheck() before trying to fire, and won't shoot if the weapon isn't ready.

DoFire ()
Use if you've already called ReadyCheck(), and want the weapon to fire without first checking if it's ready. This will cause the weapon to shoot even if it shouldn't be able to fire, such as being out of ammo.

ReadyCheck ()
Returns true if the weapon is ready to fire. Will return false if it is still cycling between shots, waiting for a reload or burst reload to finish, or if it is out of ammo.

AimCheck (***target*** : Transform, ***targetSize*** : float)
Determines if the weapon is aimed to hit the given target that has a size of targetSize. This will also take into account the weapon's aimTolerance.
This function is not currently used, but is provided in case the weapon is to be used without a turret. (The AimCheck() in ST_Turret is being used instead.)

Public Variables

shot : GameObject
The prefab of your projectile to be fired. This should have a rigidbody component in its root object.

shotSound : AudioSource
Points to the AudioSource object and will play when the gun fires.

inaccuracy : float
The error radius in degrees of shots fired.

shotSpeed : float

The velocity the shot will travel at. This value will also be given to the turret when computing an intercept path.
maxRange : float
The maximum distance the shot is to travel before expiring.
shotDuration : float
The maximum time before the shot expires.

shotSpeed, *maxRange*, and *shotDuration* are mathematically linked. You only need to supply two of the three values, and the third will be computed. If all three values are supplied, *shotDuration* will still be overwritten based on *shotSpeed* and *maxRange* to eliminate any inconsistency.

cycleTime : float
The time in seconds time between shots fired.

burstAmount : int
The number of shots fired before *burstResetTime* is triggered.

burstResetTime : float
The delay in seconds before *burstAmount* resets. The greater value of *cycleTime* and *burstResetTime* will be the delay used. This should also be less than *reloadTime*, as *reloadTime* will take precedence if they both occur on the same shot.

shotsPerRound : int
How many shots are fired per ammo point and per *Shoot()* command. This is like a shotgun blast.

maxAmmoCount : int
The maximum ammo load of the weapon. When this reaches 0, *reloadTime* begins and will replenish the ammo of the weapon.

unlimitedAmmo : boolean
The weapon never uses up ammo, but *burstAmount* and *burstResetTime* still affect the weapon.

reloadTime : float
The time in seconds it takes to refill the ammo load of the weapon.

dontReload : boolean
This weapon never replenishes its ammo when it runs out.

aimTolerance : float

Extra radius in degrees the weapon reads to the target size. This allows the weapon to fire at a target even if it’s not aimed perfectly. This can compensate for weapon inaccuracy and versus targets that change direction quickly. If negative, this will cause targeting to be more restricted to the center of the target.

exits : GunExit[] class { ***object*** : GameObject, hidden(***flare*** : GameObject), hidden(***particleComponent*** : ParticleSystem) }
Weapons may have more than one exit point. This array holds all the exit GameObjects of a single weapon. This is particularly useful for missile banks.
Drag the exit objects into the array fields. At runtime, each exit’s first child will be assumed to be the *flare* object. From this, its ParticleSystem will be stored in *particleComponent*.
Make sure the exit objects have the *flare* object as the only child object.

Hidden Variables

platformVelocity : Vector3
This is supplied by ST_TurretControl, and will only be accurate if ST_Turret *useIntercept* is true.

curAmmoCount : int
Ammo remaining.

curBurstCount : int
Shots remaining in this burst.

curExit : int
The index number of *exits* to be used for the next shot.

curInaccuracy : float
The current inaccuracy resulting from extra inaccuracy sent from ST_TurretControl because of turret rotation and elevation speed.

MF_BasicScanner

This script will scan for enemies and store them in a list. Units should be classified into factions either using tags or layers. It’s possible to use multiple scanners all providing data to the same target list.

Public Variables

targetListObject : GameObject
The location of the object holding MF_TargetList, that this scanner will provide targets for. If this is blank, it will check the same game object. If none found, then it will check the root object.

factionMethod : FactionMethodType enum { Tags, Layers }
The scanner will evaluate targets by using tags or layers. Which one to use will depend on your game.
Tags: Objects are first gathered by tag, and then individually checked for range to the scanner.
Layers: Colliders are checked by range using OverlapSphere only on layers deemed targetable. If layers are used, the object on the scan layer must have a collider.
In both cases, the root GameObject of the scanned part will be stored in *targetList*.

targetableFactions : FactionType[] enum { Side0, Side1, Side2, Side3 }
List what factions this scanner should deem as targetable. Typically, you’ll want these to be enemies of the scanning unit. Faction names must match either the tags names or the layer names.

scanRange : float
Maximum range the scanner can see.

scanInterval : float
How often in seconds the scanner refreshes the target list. Lower times result in quicker updates, but use more processing power.

MF_BasicTargeting : MF_AbstractTargeting

Searches a list of targets on MF_TargetList provided by MF_BasicScanner for the closest or furthest object. ST_TurretControl will then send that target to the ST_Turret script. Each turret may have it's own targeting script, even if they all use the same target list. As might be the case for a vehicle with multiple turrets.

Public Variables

Inherited from MF_AbstractTargeting

weaponTarget : GameObject
The current target this script has chosen. This will be gathered by ST_TurretControl.

navTarget : GameObject
Not used by SmoothTurret.

receivingObject : GameObject
An object with the script MF_AbstractPlatform. In this case, that is ST_Turret.
If blank, it looks on same object of this script. If not found, it will then check the root object. This is used to call the override method on ST_Turret.TargetWithinLimits() and skip targets that the turret can't aim at.

Native

targetListObject : GameObejct
The object containing the MF_TargetList script.
If left blank, it will first check the same GameObject as the this script. If no scanner found, it will then check the root object. If still none found, then no targets will be chosen.

priority : PriorityType { enum: Closest, Furthest }
How the turret chooses the target from the list to fire at.

keepCurrentTarget : Boolean
If true, once a target is chosen, target won't change until it no longer appears on the target list. Such as when it goes out of the scanner's range, or when it has died.

MF_TargetList

Searches a list of targets provided by MF_BasicScanner for the closest or furthest object, then sends that target to the ST_Turret script. If you have more than one turret on a single unit, you probably only need to use one target list, as each turret can choose a different target off that list.

Public Variables

targetList : Dictionary { int, TargetData }

The actual dictionary array of current targets provided by one or more scanners.

MF_BasicTargetData

The class that holds the variables for target selection, used in MF_TargetList. These values are only updated as often as the scanner provides them. If more advanced targeting routines are used, more data might be needed for MF_BasicTargeting. This class may be swapped out for something more comprehensive.

Public enums

FactionType : { Side0, Side1, Side2, Side3 }

These must match the layer or tag names that label different game factions.

FactionMethodType : { Tags, Layers }

These must match the layer or tag names that label different game factions.

Class TargetData

Public Variables

All of the variables are nullable, to accommodate the concept of “not detected”, or “unknown”, in the case that you want scanners of different capabilities.

transform : Transform

The transform of this target.

script : MF_AbstractStatus

The status script of this entry. Could be used to lookup the most current values in that script.

lastDetected : float?

The last time this entry was scanned.

sqrMagnitude : float?

The square magnitude may be used for range comparisons to avoid using Vector3.Distance(). This can be used to save performance, particularly in mobile applications.

range : float?
Range of the target.

Additional variables to be used however you see fit, without needing to replace this class. (Possibly use for health, threat, etc.)
auxValue1 : float?
auxValue2 : float?