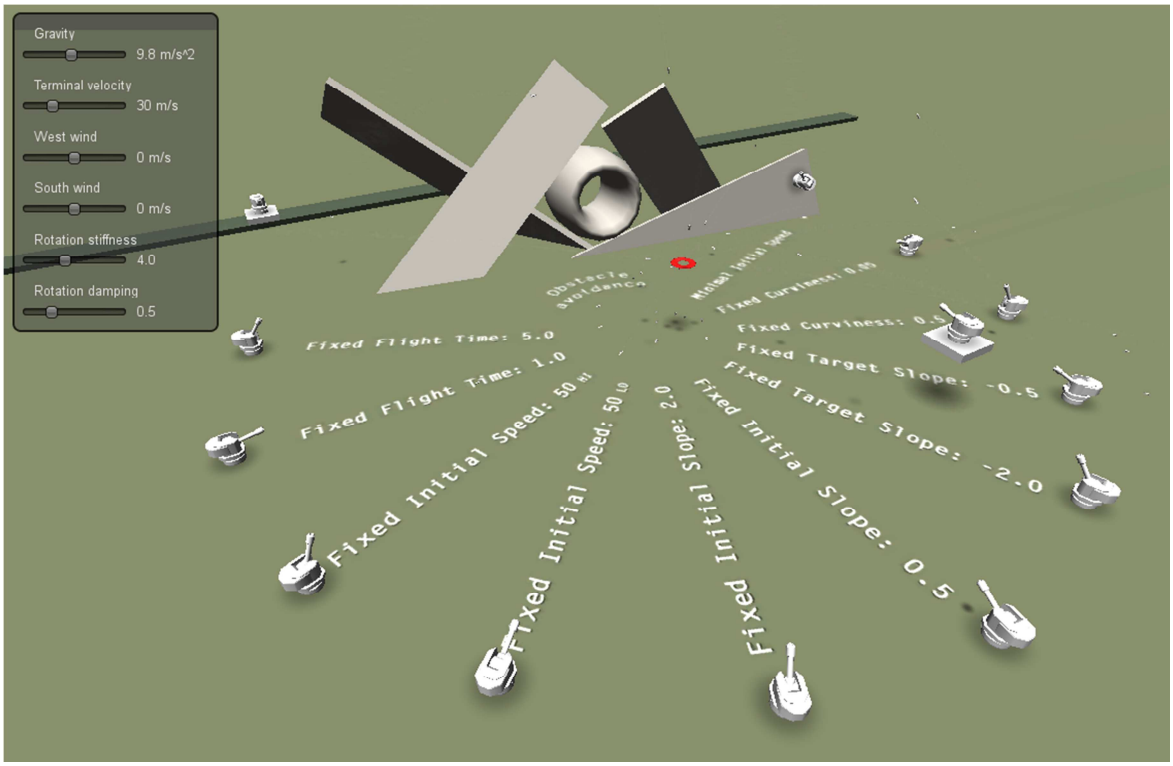
BallisticTrajectoriesUnityDemo Guide

Giliam de Carpentier, 2015-03-26
www.decarpentier.nl

The BallisticTrajectoriesUnityDemo project on BitBucket is first and foremost a practical demonstration of the work described in the paper "Analytical Ballistic Trajectories with Approximately Linear Drag". See http://www.decarpentier.nl/ballistic-trajectories for more information. This document covers some of the details of this C# demo implementation for Unity3D, which will hopefully make it easier to understand how the code can be reused for anyone's own purposes, as well as how the functions map to the formulas in the paper. This project's code and algorithms may be used freely for both commercial and non-commercial purposes provided that all BSD license conditions are met. This software comes without any support or warranty.



Controls

In the demo, the camera can be rotated by moving the mouse. To translate the camera, press either the left or right mouse button, keep it pressed, and use the W, A, S, D, Q and E keys. To move the target (i.e. the red donut), use the W, A, S and D keys without pressing the left or right mouse button.

Turret

For a gameobject to function like an aiming and shooting turret, the gameobject needs (among other things) to have an attached *Turret* script. This script does the following:

- Track the position and velocity of itself and the *Target transform* of the gameobject to hit (e.g. the red donut).
- Invoke all attached planners to come up with a launch speed and direction.
- Invoke barrel yaw and pitch updates based on this plan.
- Launch a projectile when reloading is finished and the aim is accurate enough, and trigger the recoil.

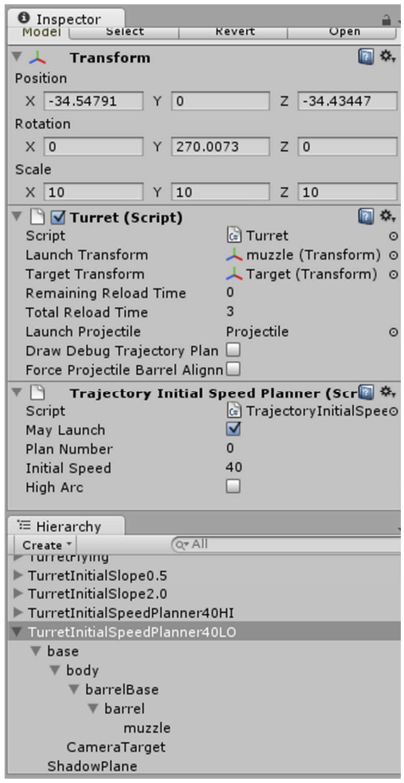


Figure 1

The *Turret* script also has a number of parameters that need to be set. See Figure 1. The *Launch transform* is the transform at which the *Launch Projectile* gameobject is spawned during launch (i.e. the tip of the muzzle). A projectile is launched only when the *Remaining Reload Time* (automatically) has counted down to zero and the barrel is (approximately) aligned with a planned launch direction. After each launch, the *Remaining Reload Time* is reset to the *Total Reload Time*. Hence, *Total Reload Time* the minimal time between projectile launches. To visualize the planned trajectory of the next projectile to be launched using 'debug lines', enable the viewport's Gizmos and enable the *Draw Debug Trajectory Plan* option.

If the option *Force Projectile Barrel Alignment* is turned on, projectiles will always be launched perfectly aligned with the barrel. This, however, can lead to loss of hit accuracy as barrel alignment is never perfect when the target moves quickly relative to the turret's maximum rotation speeds. When this option is turned off, launching is still postponed until barrel alignment is roughly correct, but projectiles will always leave the barrel in the direction of the latest plan instead of the current barrel direction, improving hit accuracy but potentially causing the projectile to leave the barrel at a slight angle.

The *Turret* script itself is kept relatively simple for the purposes of this demo, and delegates most actual functionally to other scripts. Serving mainly as an example on how to control and glue the other scripts together, it should be easy to modify, extend or make variations on. Extensions could include more complex targeting selection and more complex reload and fire behavior, for example.

Turret aiming

The *Turret* scripts looks for a *PitchMotorControl* and a *YawMotorControl* script among its (grand)child gameobjects and updates these every frame with a newly planned aiming direction if available. For aiming to work correctly, the gameobject with the *YawMotorControl* script attached needs to be a (grand)parent of the gameobject with the *PitchMotorControl* attached. For the turret shown in Figure 2, the *Turret* script has been connected to the *TurretInitialSpeedPlanner40LO* gameobject, the *YawMotorControl* script has been connected to *base*, and the *PitchMotorControl* script is connected to *barrelBase*.

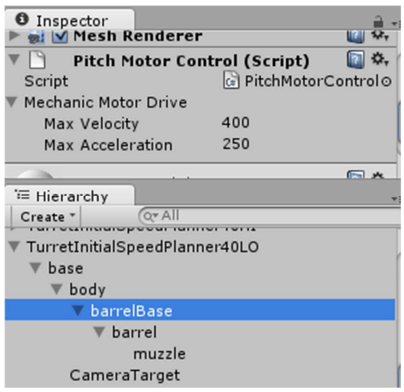


Figure 2

The *PitchMotorControl* and *YawMotorControl* script rotate around the local X and Y axis, respectively. Both rotate in such a way that the turret target direction as given by the *Turret* Script is reached as quickly as possible using the *MechanicMotorDrive* class and the specified maximum velocity and acceleration parameters. See Figure 2.

Barrel recoil

Purely as visual embellishment, the demo's turret barrels are made to run a procedural 'kickback' animation when a projectile is launched. This is achieved by attaching a *BarrelRecoil* script to the node to be animated (e.g. the barrel). The *Turret* script will look for this script among its (grand)children and automatically triggers the recoil when launching a projectile. The *BarrelRecoil* script exposes parameters to control the recoil distance, kickback speed and recovery speed. In Figure 3, the *BarrelRecoil* script connected to the *barrel* sub-gameobject.

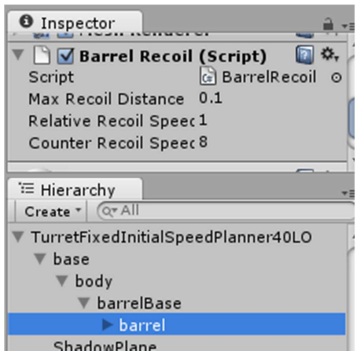


Figure 3

Projectile

A *Turret* script expects its *Launch Projectile* parameter to reference a gameobject type that has the *ProjectileKinematics* script attached to it. It uses this script to get access to the projectile's properties before the launch (which, in turn, are fed to any attached planners), and set up the position, orientation and velocity during launch. After this initial setup, the *ProjectileKinematics* script itself becomes fully responsible for updating the position and velocity each frame using the implementation of the analytical trajectory function found in the *Trajectory3D* class. Note that the trajectory does not require or use Unity's physics response functionality at all, as the trajectory is completely (pre)determined by this analytical function. As a result, the trajectory that will be followed by the projectile may differ (slightly) from one resulting from a physics simulation. See the paper's section 3.3 for more details.

The script also uses an efficient implicit integrator (which is an inherently stable simulation) to update the projectile gameobject's rotation each frame. However, when *Rotation Transform* is set to a child gameobject of the projectile, it will update the rotation of this child gameobject's transform instead. The position is always set on the gameobject that has the script attached, independent of this *Rotation Transform*. The possibility to only update the rotation of a sub-gameobject proved to be a useful feature in a previous incarnation of this project as it allowed for proper use of Unity3D's projected shadow functionality, but it's currently not used anymore for any particular purpose.

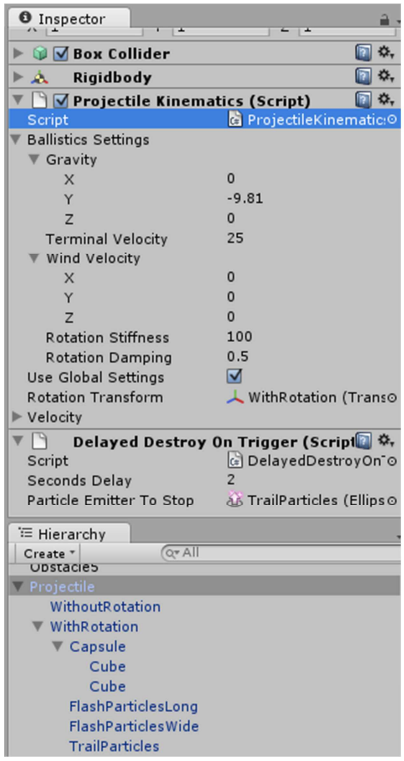


Figure 4

Another parameter exposed by the script is an instance of the *Ballistics Settings* class, which contains a gravity vector, terminal velocity, wind velocity vector and two rotation simulation parameters. See Figure 4. All of these parameters are used as input for the trajectory math (including any planners that plan the trajectory) when set before the projectile's *Start()* function is called and while *Use Global Settings* is disabled. When *Use Global Settings* is enabled, the *Ballistics Settings* member from the *Resources/GlobalSettings* 'singleton' gameobject will be used instead. In the demo, the *Use Global Settings* flag is turned on and the variables in *Resources/GlobalSettings* singleton are directly exposed as tweakable GUI sliders. These sliders are implemented in the *GlobalSettingsGUI* script, which is attached to the *MainCamera* camera gameobject.

The projectile also has a *Rigidbody* and *Box Collider* component attached to it. This is not necessary for the gameobject to function as a projectile (the *Rigidbody* has *Is Kinematic* set to true), but they are solely there to be able to detect impacts with the ground plane. Purely for demonstration purposes, the projectile also has one looping trail particle system, two one-shot muzzle flash particle systems and a *Delayed Destroy On Trigger* script attached to it. This script has nothing to do with ballistics per se, but deactivates the (updates of the) gameobject and the specified *Particle Emitter To Stop* directly after a collision has been detected, and then executes a delayed deletion of the gameobject after *Seconds Delay* seconds. As a result, the trail particle system is given enough time to fade out completely before it's destroyed together with the gameobject itself when it collides with the ground.

Planners

To launch projectiles from turret to target, the gameobject that has the *Turret* script attached to it also needs to have at least one planner script attached to it. In Figure 1, the *Trajectory Initial Speed Planner* is attached, but there are many other planner types available as well. All planners are scripts derived from the *TrajectoryPlannerBase*, have their own set of parameters, and have a *Plan Number*. The *Turret* script looks for all planners attached to its gameobject, and invokes them from low to high *Plan Number*. That allows a particular planner to fail while trying to find an immediately executable plan, and leave it to the planner with the next *Plan Number* to find an alternative. Planners with a higher *Plan Number* may also modify a plan suggested by planners with a lower *Plan Number*. Furthermore, each planner has a *May Launch* flag, which, when turned off, allows a planner to still update the barrel's target direction but disallows it to ever launch based on this plan.

Although it might be sufficient in many cases to attach only a single planner to a turret, the possibility to have multiple planners working together can be quite powerful. For example, suppose only a *Trajectory Initial Speed Planner* script is attached to a turret using *Plan Number* 0 and with *May Launch* turned on. This planner will try to find a trajectory that will start with exactly the given initial speed and which will pass through the given (future) target's position. However, when a target is too far away to be hit by a projectile with this limited initial speed, no plan will be generated and the turret won't be able to fire or aim. Now, when the target would eventually come in range, the turret would first have to rotate towards the newly found plan's launch direction and then fire. However, when a *Trajectory Minimal Initial Speed Planner* is additionally attached to the turret with *Plan Number* 1 and with *May Launch* turned off, the turret will still only fire as soon as the target comes within range of the first planner, but it no longer have to rotate first. That's because for this particular planner combination, it's per definition true that the plan based on the minimal initial speed is exactly equivalent to that based on the given initial speed when the target is at the edge of planner 0's reach. This scenario is demonstrated by the *TurretInitialSpeedPlanner40LO* turret in the demo.

Another useful example of attaching multiple planners to a single *Turret* script is to have any high-speed low-angle planner with *Plan Number* 0, and the (experimental and somewhat slow) *Trajectory Above Meshes Planner* with *Plan Number* 1. Assuming that both planners have *May Launch* enabled, the first planner continuously proposes a launch at some low-angle trajectory, while the second planner would use this trajectory as input to check if it doesn't try to shoot through any of the meshes of the game objects given in its *Obstacles* parameter. If no intersection is found, it leaves this original low-angle plan unmodified. If the trajectory would intersect, however, it will propose a higher-angle plan that goes exactly over these intersecting obstacles. This scenario is demonstrated by the *TurretOverObstacles* turret in the demo.

Each planner outputs whether it found a valid plan and (if so) the time from launch to target. The *Turret* script uses the latter to calculate the associated initial velocity using *TurretHelper's UpdatePlan()* function, which in turn calls Projectile3D.GetInitialVelocityGivenRelativeTargetAndTime().

What follows are descriptions of the individual planner types available in the demo project.

TrajectoryTimePlanner

The 'Fixed Flight Time' turrets in the demo use this planner to reach the (predicted) target position in exactly *TimeToTarget* seconds. This planner script is the most trivial planner, as it simply passes back *TimeToTarget* as its suggested plan. It never overrides or adjusts a valid plan from a planner with a lower *Plan Number*. See the paper's section 4 for the mathematical details.

TrajectoryInitialSpeedPlanner

The *TrajectoryInitialSpeedPlanner* planner script as used by the 'Fixed Initial Speed' turrets tries to find a plan that involves hitting the target position given the exact initial launch speed. When the given *Initial Speed* is great enough to even hit the target, then two solutions will be available: one for a low arc, and one for a high arc. Which one is used is determined by the *High Arc* flag. In the demo, there's one 'Fixed Initial Speed' turret with *High Arc* turned on, and one with it turned off. This planner never overrides or adjusts a valid plan from a planner with a lower *Plan Number*. See the paper's section 6.7 for the mathematical details.

TrajectoryInitialSlopePlanner

This planner plans to hit the target position while launching the projectile at the given angle. Or rather, at the given *Slope Relative To Base* slope (that is, the tangent of the local 'upwards' angle) relative to the *Slope Base Transform* transform. See the paper's section 6.3 for the mathematical details. This planner is used by the 'Fixed Initial Slope' turrets. If used on a *Turret* together with a planner that has a lower *Plan Number*, this planner guarantees that the projectile is launched at least at the *Slope Relative To Base* slope. That is, it leaves the original plan unmodified unless the earlier planner suggests launching at a slope smaller than *Slope Relative To Base*, in which case it would override the original plan.

TrajectoryTargetSlopePlanner

This planner plans to launch the projectile so that it will hit the target position at the given angle. Or rather, at the given *Target Slope* slope (that is, the tangent of the world-space 'upwards' angle). It never overrides or adjusts a valid plan from a planner with a lower *Plan Number*. See the 'Fixed Target Slope' turrets in the demo for an example. See the paper's section 6.4 for the mathematical details.

TrajectoryCurvaturePlanner

This planner plans to launch the projectile so that it will hit the target position given the *Curviness* of the trajectory from launch position to target position. The Curviness is the ratio of trajectory height and target distance. As such, higher curviness values will lead to higher trajectory arcs. It never overrides or adjusts a valid plan from a planner with a lower *Plan Number*. See the 'Fixed Curviness' turrets in the demo for an example of a high and a low *Curviness* value. See the paper's section 6.5 for the mathematical details.

TrajectoryMinimalInitialSpeedPlanner

The *TrajectoryMinimalInitialSpeed* planner plans to launch projectiles at the exact angle that leads to the smallest possible launch speed while still hitting the target position. See the 'Minimal Initial Speed' turret in the demo for an example. It never overrides or adjusts a valid plan from a planner with a lower *Plan Number*. See the paper's section 6.6 for the mathematical details.

TrajectoryAboveMeshesPlanner

This planner implements the math required to detect overlap between a given set of polygonal meshes and the trajectory planned by another planner with a lower *Plan Number*. If a collision is found, the planner modifies the previous plan to have the trajectory go over the obstacle(s). This planner can't be used on its own, as it requires another planner to come up with an initial plan first. In the demo, for example, it's used together with a *Trajectory Curvature Planner* planner script. The *TrajectoryAboveMeshesPlanner* planner is just experimental and currently not optimized for speed. The implemented math, however, is sound and covers what is briefly discussed in the paper in section 7.

"Assets\Scripts\NonBehaviours\Ballistics" folder

Most of the math is invoked from the behaviour scripts in the Assets\Scripts\Behaviour folder, but is actually implemented in non-behaviour scripts found in the Assets\Scripts\NonBehaviour folder. What follows is an overview of where the implementation of each the paper's equations and algorithms can be found in this folder.

Equation 1	Projectile3D.cs	Projectile3D.Create()
Equation 2	Projectile3D.cs	Projectile3D.Create()
Equation 3	-	-
Equation 4	Trajectory3D.cs	Trajectory3D.PositionAtTime()
Equation 5	Trajectory3D.cs	Trajectory3D.VelocityAtTime()
Equation 6	Trajectory3D.cs	Trajectory3D.GetTimeAtMaximumInDirectionN()
Equation 7	Trajectory3D.cs	Trajectory3D.GetTimeAtMaximumHeight()
Equation 8	Projectile3D.cs	Projectile3D.GetInitialVelocityGivenRelativeTargetAndTime()
Equation 9	PrincipalTrajectory.cs	PrincipalTrajectory.PositionAtTime()
Equation 10	PrincipalSpace3D.cs	PrincipalSpace3D.Create()
Equation 11	PrincipalSpace3D.cs	PrincipalSpace3D.ToPrincipalPosition()
Equation 12	PrincipalSpace3D.cs	PrincipalSpace3D.ToWorld3DPosition()
Equation 13	PrincipalSpace3D.cs	PrincipalSpace3D.ToPrincipalVelocity()
Equation 14	PrincipalSpace3D.cs	PrincipalSpace3D.ToWorld3DVelocity()
Equation 15	PrincipalTrajectory.cs	PrincipalTrajectory.PositionAtTime()
Equation 16	PrincipalTrajectory.cs	PrincipalTrajectory.PositionAtTime()
Equation 17	-	-
Equation 18	PrincipalTrajectory.cs	PrincipalTrajectory.PositionYAtX()
Equation 19	PrincipalTrajectory.cs	PrincipalTrajectory.VelocityAtTime() PrincipalTrajectory.GetTimeAtMaximumInDirectionN() PrincipalTrajectory.GetTimeAtMaximumHeight()
Equation 20	-	-
Equation 21	PrincipalProjectile.cs	PrincipalProjectile.GetInitialVelocityGivenRelativeTargetAndTime()
Equation 22	PrincipalProjectile.cs	PrincipalProjectile.GetInitialVelocityGivenRelativeTargetAndTime()
Equation 23	PrincipalTimePlanners.cs	PrincipalTimePlanners.GetTimeToTargetRGivenIntermediatePositionQ ()
Equation 24	PrincipalTimePlanners.cs	PrincipalTimePlanners.GetTimeToTargetRGivenLineToTouch()
Equation 25	PrincipalTimePlanners.cs	PrincipalTimePlanners.GetTimeToTargetRGivenInitialSlopeA()
Equation 26	PrincipalTimePlanners.cs	PrincipalTimePlanners.GetTimeToTargetRGivenTargetSlopeA()
Equation 27	PrincipalTimePlanners.cs	PrincipalTimePlanners.GetTimeToTargetRGivenArcHeightB()
Equation 28	PrincipalTimePlanners.cs	PrincipalTimePlanners.GetTimeToTargetRGivenCurvinessH()
Equation 29	PrincipalTimePlanners.cs	PrincipalTimePlanners.GetTimeToTargetRWithMinimalInitialSpeed()
Equation 30	PrincipalTimePlanners.cs	PrincipalTimePlanners.GetTimeToTargetRGivenInitialSpeedS()