One-Dimensional Ball-and-Beam Balance Simulation in Unity3D™

L. Souder II and D. Boorstein
Section 1

July 15, 2018

1 Abstract

This application note details the design and implementation of a virtual ball and beam balance system created in Unity3D™. Unity is a powerful 2D and 3D video game engine that is popular for anything from mobile games, to indie games, to triple A titles. Here we have used it and its realistic physics engine to create a control system to balance a ball at a target point on a beam. For this project we designed a PID controller in order to account for the physical properties of the ball. This application serves solely as a demo, but there are more typical uses of PID controllers in video games found in AI aim assist, steering, and any thing that aims to mimic more realistic, real world control systems.

2 Introduction

When thinking of control systems, we do not generally think of the video game space. We tend to think more along the line of robotics and industrial automation. While these are very important fields for control systems engineering, they are without a doubt not the only fields in which control systems can be utilized to a great advantage. In the world of video game design modern games are typically built on a *game engine* which is often developed separately from the game itself; whether the engine is developed by a different team of software engineers or simply purchased from an outside company. The game engine creates a layer of abstraction for the game designers. Typically it includes all of the boilerplate to deal with 3D meshes, textures, and rendering, input handling, multi-platform support, networking, and of course physics. A controls engineer in the gaming space would be mostly concerned with the physics engine. The physics engine as you may have guessed handles all of the physics of the game including gravity, force, mass, and collisions.

Now, in many cases it may seem counter-intuitive to include a control system in a

video game because the game engine exposes all aspects of every object in the game. There isn't always (or usually) a need for a controller because when you need to move an object to a certain position or apply a specific amount of force to an object, it's only a simple function call away and the game engine takes care of the rest. However, in the 21st century, with our ever growing computational resources, it has become common place to program more realistic-looking actions for objects and characters by making use of the physics engine more. Control systems have also been seen in use in the implementation of AI (artificial intelligence) for NPCs (non-player characters). This takes the form of aiming algorithms in first person shooters, steering in the racing genre, and any other game mechanic which may include a complex physics aspect. In all of these applications, it is also commonplace to design for multiple characters, scenarios, or difficulty settings. For instance, on an *easy* difficulty setting in a first person shooter, one might tune the opponents aim controller for a very high settling time, so that the opponents frequently miss their target. On a *high* difficulty setting, the opponent's aim controller would be tuned to snap to their target as quickly as possible.

Here we have designed a demo application in the Unity3D™game engine to show how a simple PID controller can be implemented using the game engine's physics API. Unity3D is a popular game engine for games on many platforms because of the level of control it gives the user while still abstracting away most of the difficult concepts. Our demo consists of the classic ball and beam control system with a behavioral script in C# which implements PID attached to the beam object. The PID controller aims to keep the ball on a target location on the beam which we have marked using a red strip on the beam. The target location can be moved from side to side using the arrow keys. We have also included a feature which allows the *player* to add a random force (parallel to the beam) to the ball in order to unsettle the system without moving the target for easy repeatability of tests.

3 Standards And Constraints

In this section, we will briefly explain the standards for our demo and the constraints that come with Unity3D and this project in general.

3.1 Standards and Regulations

For this demo, because it is entirely done in software, there are not many standards or regulations to follow other than those set by Unity itself. There are no dangers to the public presented by the simulation. There is no graphic or suggestive material (as in many games made in Unity3D) which requires us to put a rating on the demo. However, there are some simple requirements within Unity to keep everything running smoothly. For starters, anything that touches the physics engine in anyway must have a **Rigidbody**. In physics, a rigid body is object that moves as one, ideal, solid object which completely ignores deformation. In Unity a **Rigidbody** component is the same thing. It allows the programmer to give an object properties like mass, drag, shape,

and size, material properties to describe static and dynamic friction and how much an object should bounce when it collides with another object, and of course a collider. In our case, we were required to use a **Rigidbody** on both the ball and beam. Also, because the beam was supposed to be a standalone object, we were required to turn off gravity for the beam so that the entire simulation wouldn't fall away from the camera. The ball on the other hand, was completely free to move where ever it was taken, except for one require physics constraint. In order to make the ball more visible throughout the simulation, we opted to use a beam with no walls. Instead, we put a restraint on the **Rigidbody** of the ball to constrain movement to the X and Y axes only. This way, the ball was unable to roll away from or towards the screen. However, there was nothing (other than the beam controller itself) to stop the ball from rolling off the beam in the X direction. These are merely the properties put in place in order to make sure our simulated ball and beam system actually functions like a standard, physical ball and beam balance system. Essentially, the standards here were the typical ball and beam system design and the way in which Unity requires one to implement these properties.

3.2 Constraints

The most obvious constraint for this particular project (more so this particular team) was time. After realizing that there was absolutely no way to finish our previous project before the deadline, we were left with exactly 1 day to come up with a project which could be implemented with our immediately available resources, program our system, and tune our gain values to create a reasonable control system. Another unexpected constraint (which was almost certainly caused by an error on my part) was put on the angular velocity of the beam. We found that if the ball was moving to fast in relation to the beam it could bypass the collision detection and go straight through the beam. To fix this, we simply limited the angular velocity of the beam (more on this in a later section). As a side note, I later found out that this error is very likely due to how we handled the movement of the beam in software. We used the **transform.Rotate()** method from Unity's API which handles rotation directly, ignoring any physics restraints (as I type it out it seems like a much more obvious problem, but it's never that obvious in the moment). As it turns out we should have been using **gameObject.GetComponent**$<$**Rigidbody**$>$**().AddTorque()** which works directly with the physics engine to add torque to an object without ignoring other forces already at work. Also, we later found that if the beam rotated to fast, it would pop the ball into the air which creates a break in the control system since it has no control over the ball if it is not in contact with the ball. That being said, there would have had to be a restraint on the angular velocity of the beam regardless of our poorly implemented rotation.

4 Control Design

In this section we will go over the design of the system in detail. The project scene and all physical components can be seen in Figure **??**. The beam is made to ignore

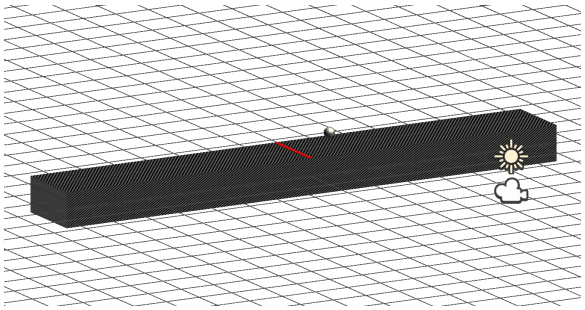

Figure 1: Project *Scene* view in Unity3D showing ball, beam, main camera, and light source

gravity, thus staying in place. The ball however is allowed to move freely on the X and Y axes. The physics engine in Unity allows the ball to behave like a realistic metal ball. It accounts for friction, gravity, momentum, and moment of inertia among other properties.

4.1 Characterizing Your System

The ball and beam system is a conceptually simple system, however, when broken down into its core properties it grows more complex. The goal of the ball and beam system is to move the ball to a target point on the beam by tilting the beam in either direction. This becomes much more complicated when you take into account the fact that the ball is affected by much more than just the angle of the beam. For one, the ball has mass and therefore has momentum. Another consideration is that the ball doesn't move at a constant speed. The presence of gravity causes the ball to accelerate quite rapidly as long as the beam is tilted.

4.2 Design Goals

The main goal of this control system was to bring the steady state error to zero so that the beam would eventually come to rest. The beam can only come to rest when the ball is centered perfectly on the target and its velocity is zero. After designing for primarily steady state error, the secondary goal was to decrease the settling time of the system to about 10 seconds or less. It is important to note that for this system, the allowable location of the target is determined by the percent overshoot of the system. If the target is near the edge of the beam and the system has a large amount of overshoot, then the ball will fall off the beam at which point it can never be centered on the target. For this reason it is very important for our percent overshoot to be very low.

4.3 Stability

The nature of this system leads it to be naturally unstable as an open loop system. Unlike a temperature control system for instance, there is no way to calibrate this system so that a certain angle of rotation on the beam will place the ball on or near a certain target every time. If the beam is left on a tilt, the ball will obviously roll off. Even if you were to time it so that a certain angle for a short time after which the beam would return to horizontal, the ball has momentum and would therefore still roll off the beam. For this reason, there is no open loop system that could come close to working for this application. To account for where the ball is in relation to where we want it to be (error signal), we would need a P controller at the very least. To counter act the momentum of the ball, we would need some kind of predictive control. This means we will need a PD controller.

4.4 Control Loop Implementation

The PID controller implemented for this system is implemented as a behavioral script attached to the beam in Unity. The beam has public members for K_p, K_i, and K_d which allows them to be easily changed from the Unity UI during the tuning phase. It also has private members for each of the error measurements including error, integral of error, and instantaneous derivative of error. The beam also has a **Transform** object for the target which is represented physically by a red strip, and the ball. This allows the beam behavioral script to access members of the target and the beam. Lastly, the beam has a private member, **zRot**, to store the rotation about the z axis to be applied to the beam in each frame.

The **Start()** function shown below is used to create a short delay to allow the ball to settle before starting the simulation. This is because of a subtle but significant quirk in Unity (and most if not all 3D game engines). If the ball was placed in the scene even *slightly* below the surface of the beam, when the simulation starts, the ball would bypass the collision detection (since it is already partially through the beam) and fall straight through the beam. For these reason, it is typical to place objects slightly above the ground (or in this case the beam). However, if the ball starts in the air, and the beam starts moving right away to correct the position of the ball, the beam would hit the ball and pop it into the air which starts a cycle where the beam is trying to over compensate while it has no control over the ball (when the ball is airborne) and therefore pops the ball into the air over and over again. The **Start()** delay was implemented in an effort to allow the ball to settle on the ground before the simulation starts. The **enabled** member is inherited from Unity's **MonoBehaviour** class. By default **enabled** is set to **true**. When **enabled** is **true**, the **Update()** block will run. When **enabled** is **false** the behavioral script will run without the **Update()** block.

```
IEnumerator Start() {
        enabled = false;
        yield return StartCoroutine (Delay());
}
```

```
IEnumerator Delay() {
        yield return new WaitForSecondsRealtime (3);
        enabled = true;
}
```

The **Update()** block is where the PID controller is actually implemented. After the 3 second delay from the **Start()** function, **enabled** will be set to true, thus allowing the **Update()** function to run. The **Update()** function runs once for every frame, and implements the control system shown in Figure 2.

```
// Update is called once per frame
void Update () {

        UpdateError ();
        Update_IError ();
        Update_dError ();

        zRot =  ((error * Kp) + (I_error * Ki) -
                (d_error * Kd)) * Time.deltaTime;
        Debug.Log (zRot);
        transform.Rotate (  new Vector3 (0, 0, -zRot),
                            Space.Self);
}
```

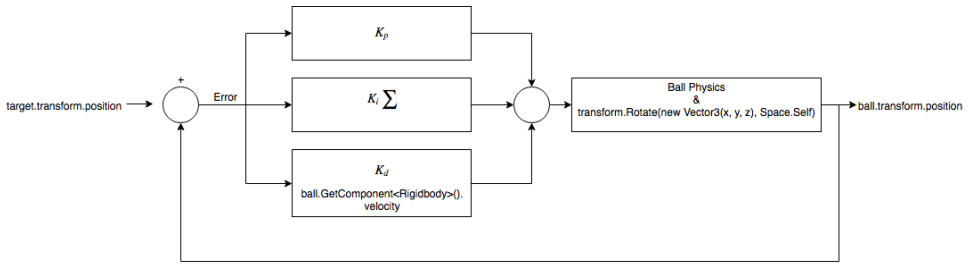

Figure 2: UML diagram of the ball and beam system

Inside the **Update()** function we will update our error measurements once per frame. This is the part of the code that implements the feedback loop by calculating the *error* node shown in Figure 2. The transfer function blocks that implement P, I, and D control are listed below. In the **UpdateError()** function, the current error is calculated by subtracting the position of the ball on the x axis from the position of the target on the x axis. It is important to note that the target is a child of the beam in the project, so its position is with respect to the beam. The ball on the other hand is *not* a child of the beam, so it operates in the world axes. Ideally, both objects would have been children of the beam, but there were unexpected errors caused by making the ball a child of the beam. At this point we decided that the error caused by this was minimal,

so we left the ball in world coordinates and accepted the error. Next, the integral of the error is calculated in the **Update_IError()** function. This function multiplies the current error by the amount of time it took to execute the last frame and adds the product to a rolling sum. Last, we calculate the instantaneous derivative of the error in the **Update_dError()** function. For this function instead of calculating the derivative, we realized that for this system, the derivative of the error had a physical manifestation. In this case, the derivative of the error represented the velocity of the ball. Because we are using Unity for this simulation, we have direct access to the velocity of the ball through its **Rigidbody** component.

```
void UpdateError(){
        error = target.position.x - ball.position.x;
        Debug.Log("ErrorX: " + error);
}

void Update_IError() {
        I_error += (error * Time.deltaTime);
}

void Update_dError (){
        d_error = ball.GetComponent<Rigidbody>
            ().velocity.x;
```

The last thing done in the **Update()** function is the actual rotation of the beam. First, the rotation about the z axis, **zRot** is calculated by summing all of the error measurements multiplied by their respective gain constants. This is the implementation of the rightmost summing node in Figure 2. Then the rotation is scaled by the time of the last frame and the beam is rotated.

5 Design Discussion

Implementing this system in Unity3D consisted of three steps.

1. Creating and arranging all objects in the scene
2. Writing scripts to define object behavior (implementing the PID controller)
3. Adjusting gain values to tune the system

First we created out ball and beam objects in the scene view. The beam was a simple cube 3D object that we scaled on the x axis to get an elongated rectangular prism. The ball was sphere 3D object from the Unity prefabs. The ball was given a **Rigidbody** component so that we could access properties such as velocity and add forces. The beam was also given a **Rigidbody** component in order to interact with the collision detection system built into the physics engine, however, the beam was given a kinematic **Rigidbody** which means it is unaffected by forces from other objects. We also disabled gravity on the beam so that the simulation would not fall away at the start

of the simulation. This is a common step to take in creating platforms and obstacles in games. The target was created as another scaled cube 3D object. The **isTrigger** box was checked on the target's collider so that it would not interfere with other objects. Essentially, a trigger object is an object that will allow other objects to pass through it. Next, all of the objects were given textures so that they could be easily distinguished. The target was placed inside the beam with a small piece of it sticking out from the top of the beam, so that only a thin red strip was visible on the beam. Next, the ball was placed on top of the beam with the bottom of the ball slightly above the beam, so that the ball would not fall through the beam at run time. The ball's **Rigidbody** component was then restricted to move only in the x and y axes. This kept it from rolling off the beam in the z direction without the need for walls on the beam which would obstruct the view. Lastly, we set the camera in a position slightly above the beam, looking down on a very small angle below the horizontal. We placed a point light above the camera to light the scene.

Next we implemented the control system as a script component on the beam as shown in the sections above. The system implemented can be summed up into the following block diagram.

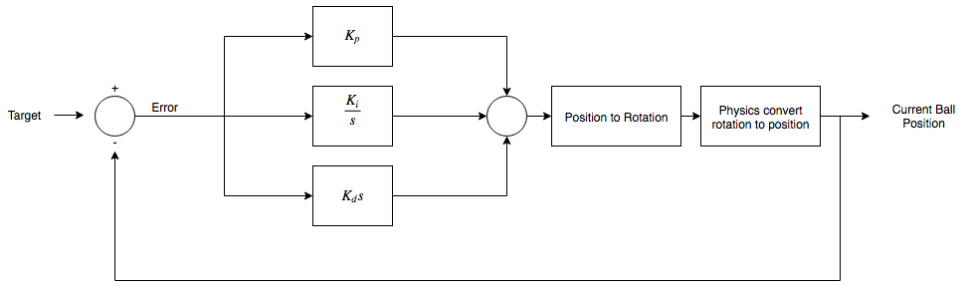

Figure 3: Block diagram of the ball and beam system

The blocks in parallel are the blocks that are implemented in the beam script to implement the PID controller. These blocks are essentially taking the position error, integral of the error, and instantaneous derivative of the error and multiplying them by their respective gain constants to determine the change in rotation that should be applied in the next frame. The block labeled "Position to Rotation" represents the **transform.Rotate()** function. In this case, we are suing the output of the right most summing node as the rotation about the z axis. The last block represents the engine physics which then move the ball on the x axis based on the angle of the beam, gravity, momentum, mass of the ball, friction, and drag. There is no known transfer function for these last two blocks because they are done in the background by the game engine.

The last thing we did was tune our controller. We started by setting K_i and K_d to 0, so that we could tune only K_p. We started K_p around 2 and increased it until we obtained a system in which the ball would oscillate from side to side. Given some

aforementioned errors (unknown at the time) in our implementation of the rotation, this was particularly difficult because oscillation only occurred at a relatively high gain at which point the ball had a tendency to go through the beam. After getting the system to a state of oscillation, we then tuned K_d to account for the overshoot of our system. K_d was increased until the overshoot was decreased significantly. The derivative control was used to slow down the ball as it approached the center of the beam. This was particularly important for our system because if the overshoot was too high the ball could roll off either end of the beam. This is especially problematic when the target is set close too the edge of the beam. Finally, we tuned K_i to improve steady state error. Without integral control, the system would inevitably oscillate about the target point without ever settling. K_i was increased in order to counteract the oscillation. The determined gain constants are given in Table 1.

Table 1: Gain constants determined by empirical methods

Gain Constant	Value
K_p	60
K_i	30
K_d	120

6 Results and Conclusions

Our system was designed to move the ball to a chosen target location from anywhere on the beam. That being said, the response of the system varies slightly based on the distance the balls starts from the target and how fast the ball is moving at the start of the simulation. However, we have determined through a series of trials, that our system will always come to a steady state, but the time it takes to reach a perfectly steady state varies. On average, the ball will reach a state in which it stays on the marker (it may still be oscillating very slightly at this point as long as it doesn't leave the target marker) in about 10 seconds. The overshoot of the system has shown to be anywhere from 0.3 to 0.6 meters (assuming the default units in Unity are in fact meters). This is small enough that be target is able to be placed very close to the edge without the ball falling off, with the exception of some physics glitches most likely caused by the our implementation's inability to work properly with the collision detection system in Unity3D. This control system could very easily be modified for other systems in video games such as aim assist and steering for artificial intelligence in non-player characters. The main difference would be that the actuator would change from the rotation of our beam to some other physical property. As mentioned before, because we allowed our gain constants to be configurable from the Unity UI, it would be very easy for someone to tune multiple controllers for various settings using this same script. In the way of improvements for this controller, the biggest change would simply be to change the actuator from a call to **transform.Rotate()** to **gameObject.GetComponent<Rigidbody>().AddTorque()**. This would allow the beam to better interact with the physics engine and most importantly the collision detector of the game engine which should fix some of the glitches we were seeing and

allow us to tune the system better without worrying about said glitches. Also, it would be worth while to add a conditional piece to the control system so that if the ball is not in contact with the beam, the beam will pause until the ball comes back to the beam. This should help with the case where the beam over corrects and pops the ball into the air thus removing a nonlinearity in the system.

Finally, to see this demo again, you can visit `https://github.com/Llcoolsouder/Unity_PID`. This repository includes the entire Unity3D project, so that you can build the demo on your own system, or modify the demo to meet your own needs. It also includes the binaries built on my personal computer. The README for this repository includes, brief instructions on how to use the code in this project as well as the demo itself.

6.1 Work Distribution

Lonnie Souder II

- Scene setup
- C# scripting for Unity PID implementation
- C# scripting for player input
- Github repository maintenance
- Debugging and tuning

Damon Boorstein

- Debugging
- PID gain tuning
- Research methods and bugs

7 Appendix A

7.1 User Input Scripts

7.1.1 MoveTarget.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MoveTarget : MonoBehaviour {

        float xMove = 0;

        // Update is called once per frame
        void Update () {
                xMove = Input.GetAxis ("Horizontal") *
                        Time.deltaTime;
                transform.localPosition +=
                        new Vector3 (xMove, 0, 0);
        }
}
```

7.1.2 BallControl.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BallControl : MonoBehaviour {

        public Transform beam;

        // Update is called once per frame
        void Update () {
                if (Input.GetKeyDown (KeyCode.Space)) {
                        float dir =
                            Random.Range (-10f, 10f);
                        gameObject.GetComponent<Rigidbody>
                            ().AddForce
                            (beam.transform.right * dir,
                            ForceMode.Impulse);
                }
        }
}
```