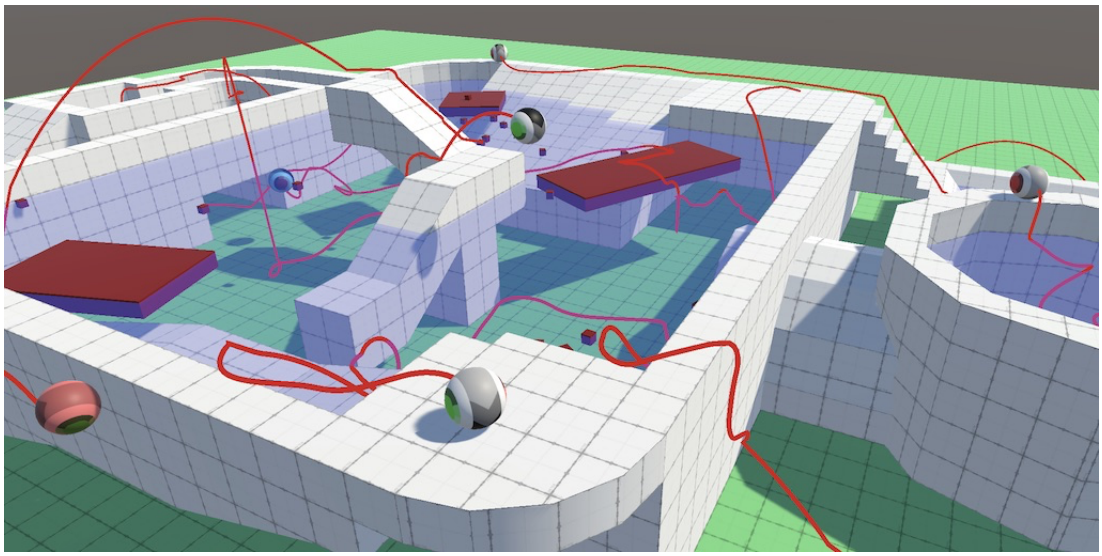
published 2020-08-30

Rolling Animated Sphere

Accelerate based on current velocity.
Make the sphere visually roll.
Align the sphere with its motion.
Stay aligned with the ground, even when it moves.

This is the 11th and final installment of a tutorial series about controlling the movement of a character. It turns our featureless sphere into a rolling ball.

This tutorial is made with Unity 2019.4.8f1. It also uses the ProBuilder package.

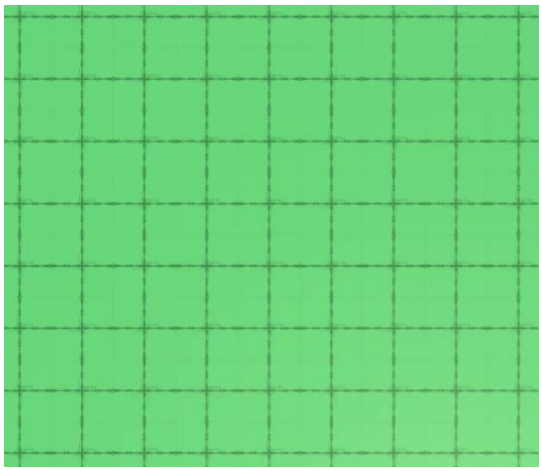


Rolling around at the playground.

Velocity-Relative Acceleration

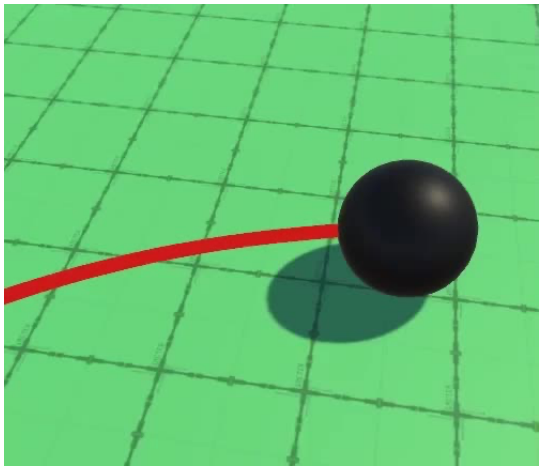
Our current acceleration approach is relative to the player's input space, for which we use either world space or the orbit camera. This works fine, but it ignores the current velocity of the sphere when applying acceleration. This can become obvious when letting go of the controls while not moving straight or diagonally aligned with the X and Z control axes. If acceleration isn't sufficient for a near instantaneous stop velocity will align itself with the nearest axis. This happens because the sphere decelerates at the same rate along both axes, so the smallest component reaches zero first.

This is most obvious when using keys to control the sphere, instead of a stick. To eliminate input delay I increased the *Gravity* and *Sensitivity* of the *Horizontal* and *Vertical* input keys from 3 to 3000.



Axial bias.

The same phenomenon causes sharp zigzag motion when moving left and right when controlling the sphere with an aligning orbit camera.



Sharp zigzag.

Although the current control method is biased it can be very snappy, so you might not want to change it. But let's go ahead and remove the bias.

1.1 Clamping Velocity Delta

To remove the bias we have to make the velocity adjustment of all dimensions interdependent. So we'll switch to working with an adjustment vector instead of isolated old and new values. While we're at it, let's also swap the components of sideways and vertical movement, so Y is up-down in `MovingSphere.Update`.

```
		playerInput.x = Input.GetAxis("Horizontal");
		playerInput.z = Input.GetAxis("Vertical");
		playerInput.y = Swimming ? Input.GetAxis("UpDown") : 0f;
```

Next, remove the current X and Z values from `AdjustVelocity`, replacing them with an adjustment vector where we directly calculate the desired velocity adjustment along X and Z.

```
		Vector3 relativeVelocity = velocity - connectionVelocity;
		//float currentX = Vector3.Dot(relativeVelocity, xAxis);
		//float currentZ = Vector3.Dot(relativeVelocity, zAxis);

		Vector3 adjustment;
		adjustment.x =
			playerInput.x * speed - Vector3.Dot(relativeVelocity, xAxis);
		adjustment.z =
			playerInput.z * speed - Vector3.Dot(relativeVelocity, zAxis);
```

Also include the Y adjustment at this point, if we're swimming. Otherwise it's zero.

```
		adjustment.z =
			playerInput.z * speed - Vector3.Dot(relativeVelocity, zAxis);
		adjustment.y = Swimming ?
			playerInput.y * speed - Vector3.Dot(relativeVelocity, upAxis) : 0f;
```

Then instead of calculating new values for X and Z independently we clamp the adjustment vector by the maximum speed change. This applies acceleration once and removes the bias.

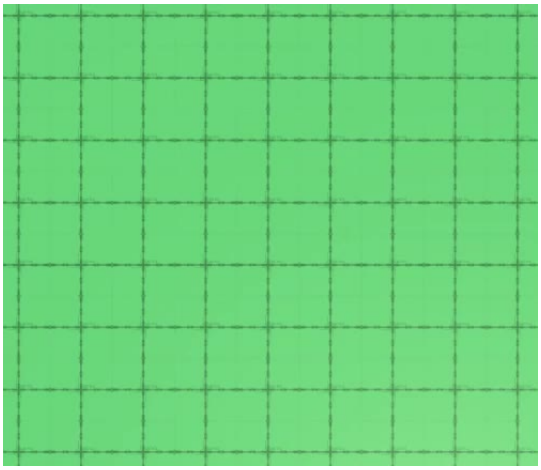
```
		//float maxSpeedChange = acceleration * Time.deltaTime;

		//float newX =
		//	Mathf.MoveTowards(currentX, playerInput.x * speed, maxSpeedChange);
		//float newZ =
			//Mathf.MoveTowards(currentZ, playerInput.y * speed, maxSpeedChange);

		adjustment =
			Vector3.ClampMagnitude(adjustment, acceleration * Time.deltaTime);
```

The velocity change is now the X plus Z axes scaled by their respective adjustments.

```
		velocity += xAxis * adjustment.x + zAxis * adjustment.z;
```

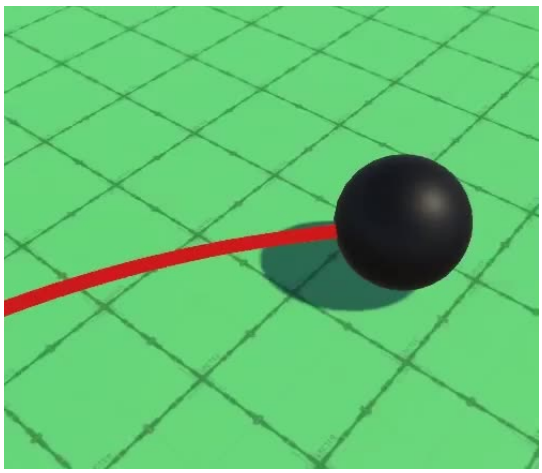
Plus the adjustment along the Y axis, if needed.

```
		if (Swimming) {
			//float currentY = Vector3.Dot(relativeVelocity, upAxis);
			//float newY = Mathf.MoveTowards(
			//	currentY, playerInput.z * speed, maxSpeedChange
			//);
			velocity += upAxis * adjustment.y;
		}
```



No Axial Bias.

This new approach also replaces the sharp sideways zigzag motion with a smooth curve. This is more realistic, as it makes turning at higher velocities more difficult, but it also makes control less precise. You can compensate for this by increasing max acceleration.



Smooth zigzag.

2 Rolling Ball

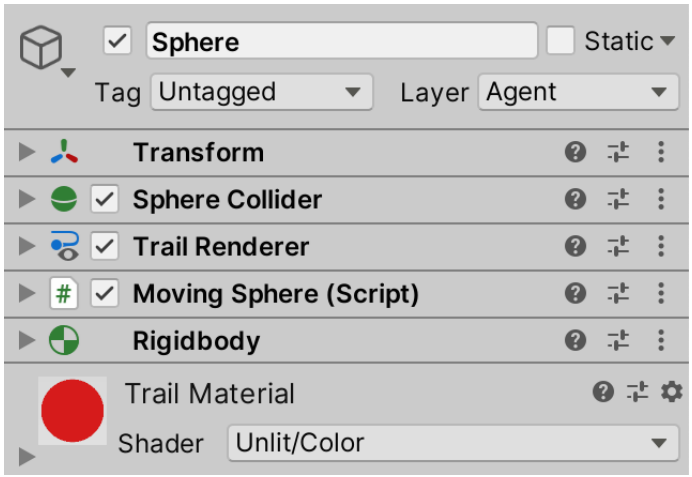
Our sphere moves by sliding around on surfaces, jumping, swimming, and falling. As long as the sphere has a uniform color it looks the same from any orientation and thus we wouldn't be able to see whether it rolls or slides. To get a better sense of the sphere's motion we're going to make it roll.

2.1 Ball Child

To make rolling obvious we need to apply a texture to the sphere. Here is a texture for this purpose. It's a 512×256 texture designed to be wrapped around a sphere, with an arrow or track-like stripe in the middle and red-green coloration for its left and right sides. Apply it to the sphere materials that we have and set the albedo of the normal material to white.

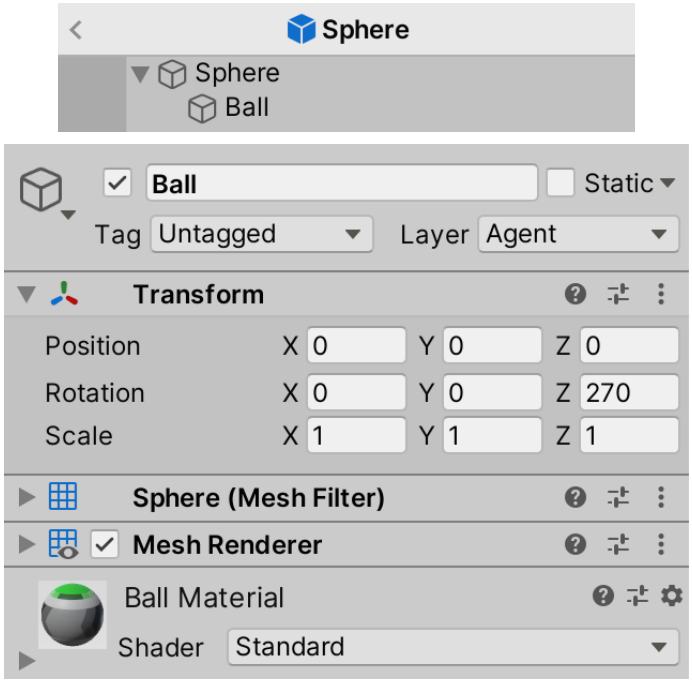
Ball texture.

The moving sphere itself doesn't rotate, we'll give it a ball child object instead. Begin by removing the mesh renderer and filter components from the sphere prefab.



Sphere prefab components.

Then add a ball child object to it, which is a default sphere with its collider removed. The default sphere mesh is a cube sphere with default sphere UV coordinates, so the texture is subject to heavy distortion at the poles. For this reason we rotate the ball by 270° around the Z axis, putting the poles on the sides, which matches the uniformly-colored red and green regions of the texture.

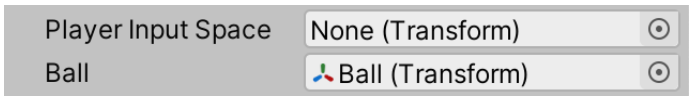


Ball child object.

2.2 Adjusting Ball Materials

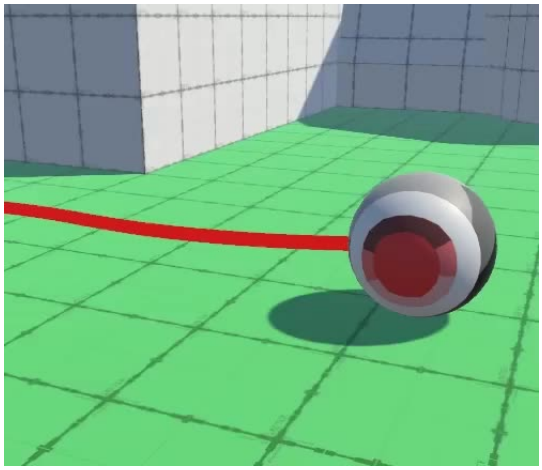
From now on we have to change the ball's material instead of the sphere's. Add a configuration option for the ball's `Transform` component to `MovingSphere` and get its `MeshRenderer` in `Awake`. Then hook up the reference in the prefab.

```
	[SerializeField]
	Transform playerInputSpace = default, ball = default;

	…

	void Awake () {
		body = GetComponent<Rigidbody>();
		body.useGravity = false;
		meshRenderer = ball.GetComponent<MeshRenderer>();
		OnValidate();
	}
```



Prefab with reference to its ball.

At this point we have a textured ball, which makes it obvious that it slides.



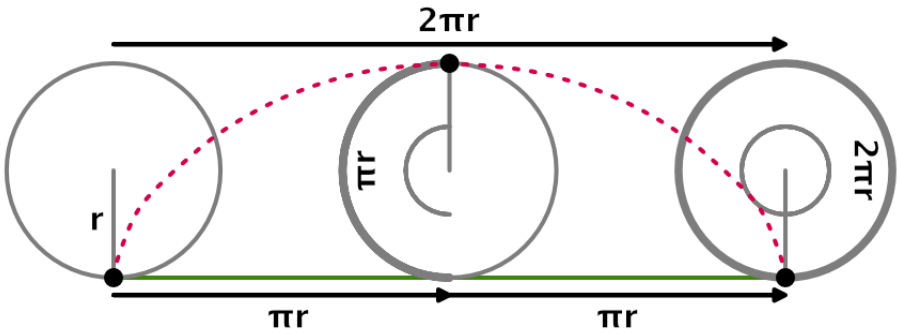
Sliding Ball.

Let's put all the code related to updating the ball in a separate `UpdateBall` method. Move the material-setting code there. Also, switch to using conditional blocks, as we'll make more changes based on the movement mode later.

```
	void Update () {
		…

		//meshRenderer.material =
			//Climbing ? climbingMaterial :
			//Swimming ? swimmingMaterial : normalMaterial;
		UpdateBall();
	}

	void UpdateBall () {
		Material ballMaterial = normalMaterial;
		if (Climbing) {
			ballMaterial = climbingMaterial;
		}
		else if (Swimming) {
			ballMaterial = swimmingMaterial;
		}
		meshRenderer.material = ballMaterial;
	}
```

2.3 Motion

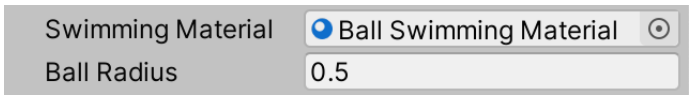
To make the ball roll we have to rotate it so its surface movement matches its motion. The simplest perfect case is a ball rolling in a straight line. Only a single point of the ball's surface touches the ground at each moment. As the sphere moves forward it rotates, and once it has completed a full rotation of 360° the same point touches the ground again. During that time the point traveled in a circle relative to the ball's origin. Thus the distance traveled equals the circumference of that circle, which is 2π times the radius of the ball.



Rolling and rotating.

This means that we need to know the radius of the ball, which depends on the sphere's size. Let's add a configuration option for it, which must be positive, set to 0.5 by default, matching the default sphere.

```
	[SerializeField, Min(0.1f)]
	float ballRadius = 0.5f;
```



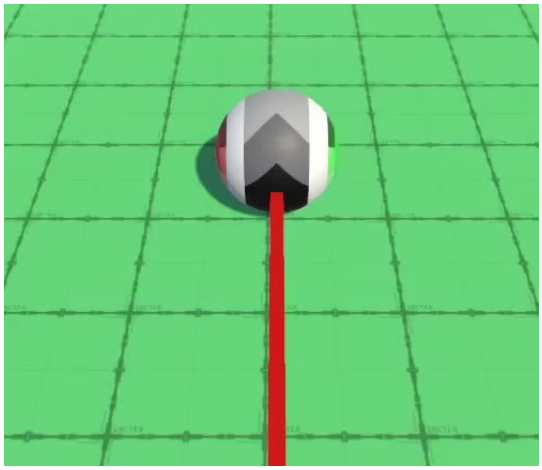
Ball radius.

We make the ball roll during the regular per-frame update—in `UpdateBall`—because it's purely a visual effect. But the sphere moves during the physics steps, so we might end up with linear motion in between if the frame rate is high enough. That's fine as long as the sphere's `Rigidbody` is set to interpolate. We can then find the appropriate movement vector by dividing the body's velocity by the time delta. The covered distance is the magnitude of that vector. This isn't perfect but it's good enough visually.

```
	void UpdateBall () {
		…

		Vector3 movement = body.velocity * Time.deltaTime;
		float distance = movement.magnitude;
	}
```

The corresponding rotation angle is then the distance times 180 divided by π, then divided by the radius. To make the ball roll we create a rotation of that angle via `Quaternion.Euler`, multiply that with the ball's rotation, and use that as its new rotation. Initially we'll use the world X axis for the rotation axis.

```
        float distance = movement.magnitude;
        float angle = distance * (180f / Mathf.PI) / ballRadius;
        ball.localRotation =
            Quaternion.Euler(Vector3.right * angle) * ball.localRotation;
```



Rotating around fixed axis.

2.4 Rotation Axis

This approach works, as long as we're moving forward along the world Z axis. To make it work for any direction we have to derive the rotation axis from the movement direction and the contact normal. As the contact normal gets cleared to zero each physics step we'll have to keep track of the last one. Copy the normal to a field just before we clear it.

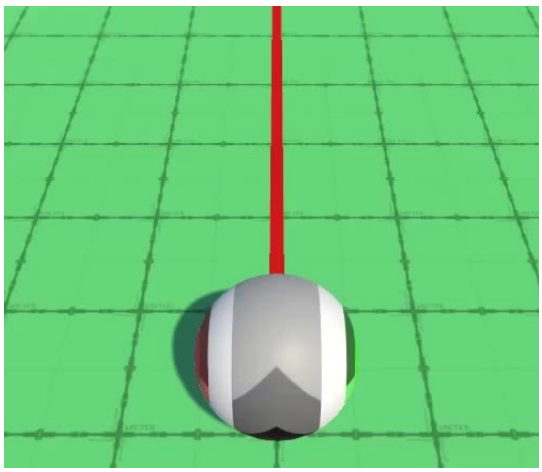
```
	Vector3 lastContactNormal;

	…

	void ClearState () {
		lastContactNormal = contactNormal;
		…
	}
```

Now we can find the rotation axis in `UpdateBall` by taking the cross product of the last contact normal and the movement vector, normalizing the result.

```
		Vector3 rotationAxis =
			Vector3.Cross(lastContactNormal, movement).normalized;
		ball.localRotation =
			Quaternion.Euler(rotationAxis * angle) * ball.localRotation;
```

However, this won't work when standing still, so abort if there is very little movement that frame, say less than 0.001.

```
		float distance = movement.magnitude;
		if (distance < 0.001f) {
			return;
		}
```

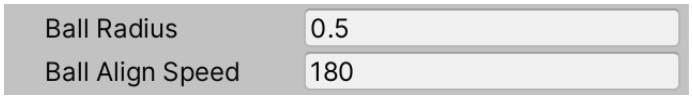


Rolling in the appropriate direction.

2.5 Aligning the Ball

The ball now rotates correctly, but a consequence of this is that its texture can end up with an arbitrary orientation. As its pattern has an implied direction let's make the ball align itself with its forward movement. This requires an additional rotation that gets applied on top of rolling. How quickly it aligns itself can be made configurable, just like the align speed of the orbit camera, so add an option for that.

```
	[SerializeField, Min(0f)]
	float ballAlignSpeed = 180f;
```



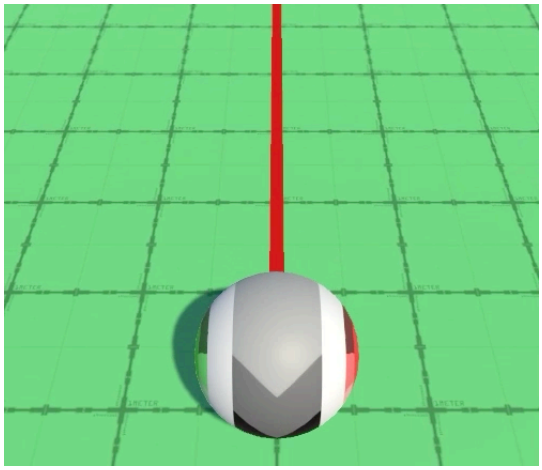
Ball align speed set to 180°.

Copy `OrbitCamera.UpdateGravityAlignment` to `MovingSphere`, rename it to `AlignBallRotation` and adjust it so it works for the ball. Give it two parameters, first the rotation axis and second the ball's rotation. Replace the gravity alignment with the ball's local up axis and gravity with the rotation axis. Finally, apply the adjustment to the ball's rotation and return it.

```
	Quaternion AlignBallRotation (Vector3 rotationAxis, Quaternion rotation) {
		Vector3 ballAxis = ball.up;
		float dot = Mathf.Clamp(Vector3.Dot(ballAxis, rotationAxis), -1f, 1f);
		float angle = Mathf.Acos(dot) * Mathf.Rad2Deg;
		float maxAngle = ballAlignSpeed * Time.deltaTime;

		Quaternion newAlignment =
			Quaternion.FromToRotation(ballAxis, rotationAxis) * rotation;
		if (angle <= maxAngle) {
			return newAlignment;
		}
		else {
			return Quaternion.SlerpUnclamped(
				rotation, newAlignment, maxAngle / angle
			);
		}
	}
```

Invoke the method in `UpdateBall` if the align speed is positive.

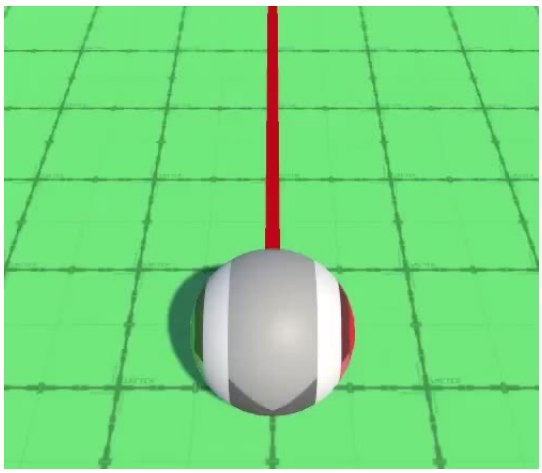
```
		Vector3 rotationAxis =
			Vector3.Cross(lastContactNormal, movement).normalized;
		Quaternion rotation =
			Quaternion.Euler(rotationAxis * angle) * ball.localRotation;
		if (ballAlignSpeed > 0f) {
			rotation = AlignBallRotation(rotationAxis, rotation);
		}
		ball.localRotation = rotation;
```



Rolling forward.

This works, but it makes more sense if the alignment is based on traveled distance rather than time. That way the alignment slows down and speeds up along with motion. So pass the distance to `AlignBallRotation` and use it instead of the time delta. Thus the configured speed is in degrees per traveled unit, rather than per second.

```
	void UpdateBall () {
		…
		if (ballAlignSpeed > 0f) {
			rotation = AlignBallRotation(rotationAxis, rotation, distance);
		}
		ball.localRotation = rotation;
	}

	Quaternion AlignBallRotation (
		Vector3 rotationAxis, Quaternion rotation, float traveledDistance
	) {
		Vector3 ballAxis = ball.up;
		float dot = Mathf.Clamp(Vector3.Dot(ballAxis, rotationAxis), -1f, 1f);
		float angle = Mathf.Acos(dot) * Mathf.Rad2Deg;
		float maxAngle = ballAlignSpeed * traveledDistance;

		…
	}
```



Distance-based alignment, ball align speed 45°.

Could the ball keep the same orientation when reversing?

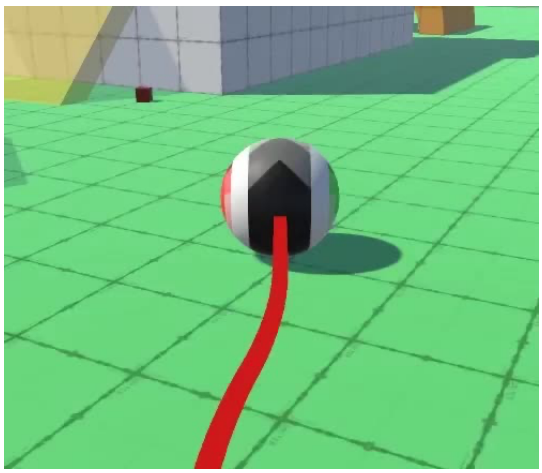
Yes. You can do that by checking whether the alignment angle is greater than 90°. If so, reduce the angle by 90° and negate the rotation axis before aligning.

3 Rolling in Context

Our sphere now rolls appropriately in simple cases, but there are some special cases that we have to consider to make it behave well in general.

3.1 Steep Slopes

As we use the last contact normal to derive the rotation axis, when in the air the balls roll as if it were on flat ground. This happens even when the ball slides along a wall.



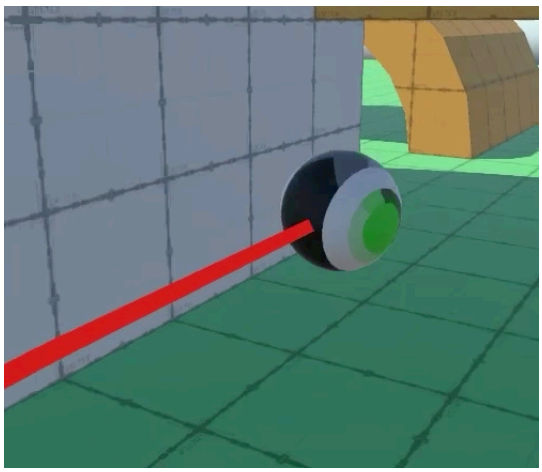
Sliding past a wall.

Although this is correct, it would look more interesting if the sphere would align itself so it rolls along the wall's surface. This would also provide a hint that it's possible to jump off the wall. To make this possible we also have to keep track of the last steep normal.

```
	Vector3 lastContactNormal, lastSteepNormal;

	…

	void ClearState () {
		lastContactNormal = contactNormal;
		lastSteepNormal = steepNormal;
		…
	}
```

Now we can end up using different normals to determine our rotation plane in `UpdateBall`. The default is to use the last contact normal, but if we're not climbing nor swimming, are not on the ground, but are on a steep surface, then use the last steep normal instead.

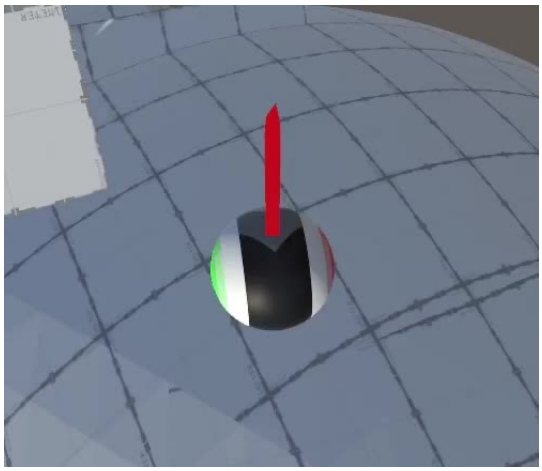
```
		Vector3 rotationPlaneNormal = lastContactNormal;
		if (Climbing) {
			ballMaterial = climbingMaterial;
		}
		else if (Swimming) {
			ballMaterial = swimmingMaterial;
		}
		else if (!OnGround) {
			if (OnSteep) {
				rotationPlaneNormal = lastSteepNormal;
			}
		}
		…
		Vector3 rotationAxis =
			Vector3.Cross(rotationPlaneNormal, movement / distance);
```



Rolling along a wall.

3.2 Ignoring Upward Motion

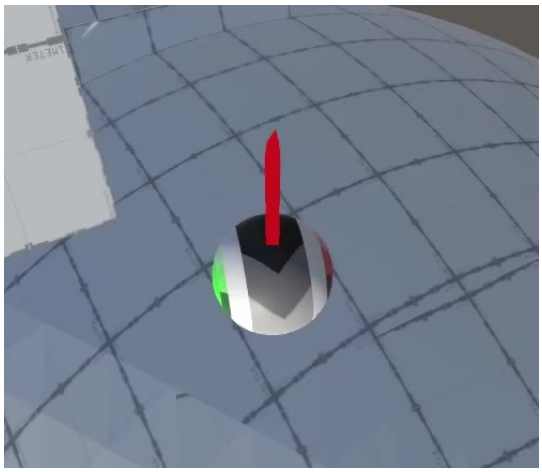
Currently we use movement in all three dimensions to determine the ball's rotation and alignment. This means that relative upward and downward motion affects it. Unfortunately this can cause trouble when jumping straight up, especially in complex situations where gravity isn't uniform. Jumping straight up can result in small jittery motion, which leads to erratic ball rotation.



Unstable jumping.

We can reduce this effect by ignoring the relative vertical motion when updating the ball, which is done by projecting the movement on the rotation plane normal and subtracting that from the vector.

```
		Vector3 movement = body.velocity * Time.deltaTime;
		movement -=
			rotationPlaneNormal * Vector3.Dot(movement, rotationPlaneNormal);
```



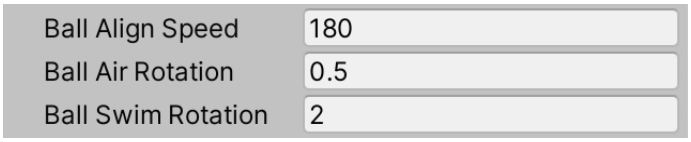
Stable jumping.

3.3 Air and Swim Rotation

If makes sense that the ball rolls when moving along a surface, but when in the air or swimming it technically doesn't need to roll. However, as our sphere is self-propelled it's intuitive that it always rolls. But when not in direct contact with a surface its rotation has no surface to match, so we could make it rotate at a different speed.

Add separate configuration options for the ball's air and swim rotation. At minimum the speeds could be zero. Let's set the air rotation to 0.5 by default, which makes the ball rotate slower while in the air. And let's use 2 for the default swim rotation factor, so the ball appears to work harder while swimming.

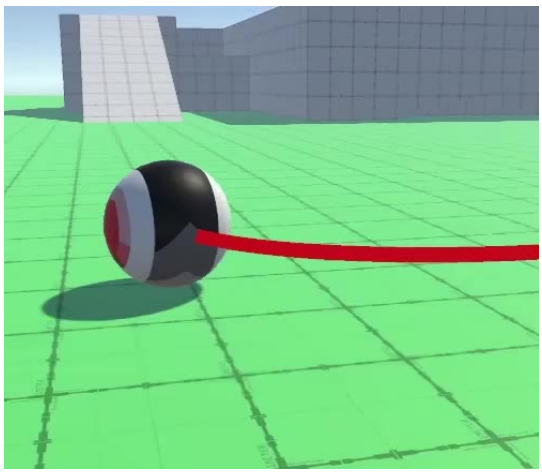
```
	[SerializeField, Min(0f)]
	float
		ballAirRotation = 0.5f,
		ballSwimRotation = 2f;
```



Air and swim rotation factors.

We adjust the rotation speed by scaling the angle by a rotation factor in `UpdateBall`. It's 1 by default, but we should use the appropriate configured speed when either swimming or when not in contact with anything.

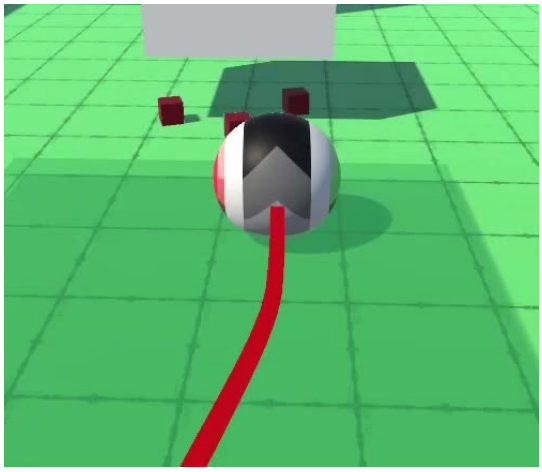
```
		float rotationFactor = 1f;
		if (Climbing) {
			ballMaterial = climbingMaterial;
		}
		else if (Swimming) {
			ballMaterial = swimmingMaterial;
			rotationFactor = ballSwimRotation;
		}
		else if (!OnGround) {
			if (OnSteep) {
				rotationPlaneNormal = lastSteepNormal;
			}
			else {
				rotationFactor = ballAirRotation;
			}
		}
		…
		float angle = distance * rotationFactor * (180f / Mathf.PI) / ballRadius;
```



Different roll speeds.

3.4 Rolling on Moving Surfaces

The last step of making our sphere roll correctly is to make it work in combination with moving surfaces. Right now the ball inherits the movement of the connected body, which makes it rotate inappropriately.



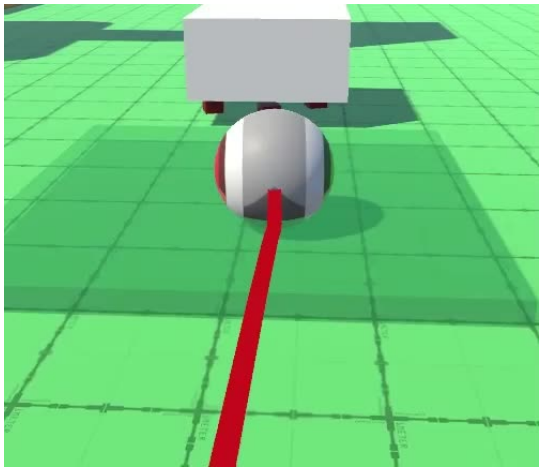
Rolling due to platform motion.

To compensate for the surface motion we have to keep track of the last connection velocity.

```
	Vector3 lastContactNormal, lastSteepNormal, lastConnectionVelocity;

	…

	void ClearState () {
		lastContactNormal = contactNormal;
		lastSteepNormal = steepNormal;
		lastConnectionVelocity = connectionVelocity;
		…
	}
```

Then we can subtract is from the body's velocity in `UpdateBall`.

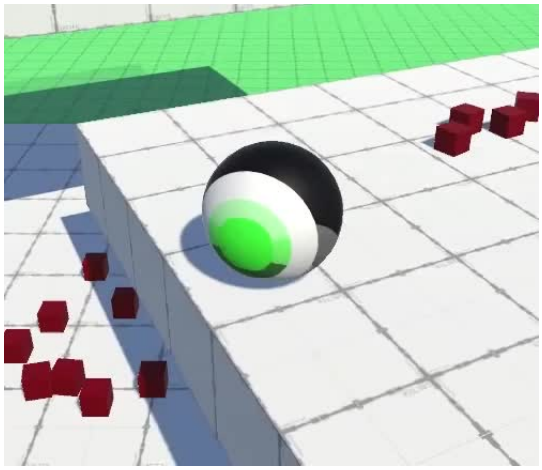
```
		Vector3 movement =
			(body.velocity - lastConnectionVelocity) * Time.deltaTime;
```



Relative rolling.

3.5 Rotating Along With Surfaces

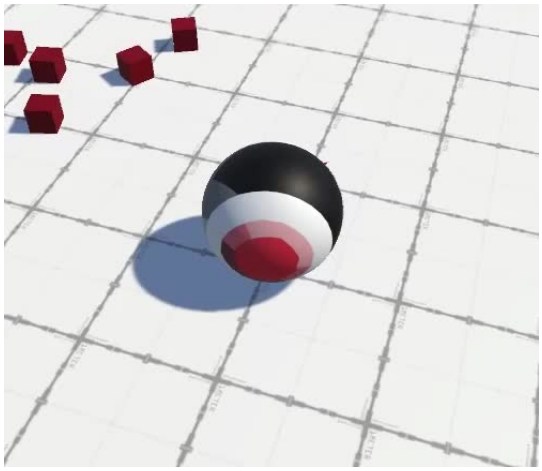
Besides moving, the connected body could also rotate. We take this into consideration when determining movement, but the sphere's alignment is not yet affected by it.



Ignoring platform rotation.

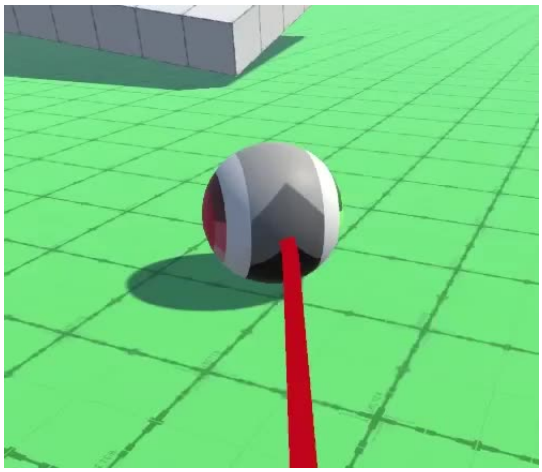
In this case we can make the ball rotate along with the platform by creating a rotation from the connected body's angular velocity, scaled by the time delta. We can use the `Rigidbody.angularVelocity` property for this, which is in radians to we have to multiply it with `Mathf.Rad2Deg` before we pass it to `Quaternion.Euler`. Multiply this rotation with the ball's current rotation before rolling it. We only have to do this when we're staying connected to a body, but if so also when the ball is standing still.

```
        Quaternion rotation = ball.localRotation;
        if (connectedBody && connectedBody == previousConnectedBody) {
            rotation = Quaternion.Euler(
                connectedBody.angularVelocity * (Mathf.Rad2Deg * Time.deltaTime)
            ) * rotation;
            if (distance < 0.001f) {
                ball.localRotation = rotation;
                return;
            }
        }
        else if (distance < 0.001f) {
            return;
        }

        float angle = distance * rotationFactor * (180f / Mathf.PI) / ballRadius;
        Vector3 rotationAxis =
            Vector3.Cross(rotationPlaneNormal, movement).normalized;
        //Quaternion rotation =
        rotation = Quaternion.Euler(rotationAxis * angle) * rotation;
```



Rotating along with platform.

As this is a 3D rotation it makes the ball inherit any rotation of the connected body. So if the surface wobbles the ball will wobble along with it.



Rolling on a wobbling platform.

This concludes the Movement series. The next step from here would be to replace the ball with a more complex body, like a human. That's the subject of a follow-up series that I'll create in the future.

license

repository

Enjoying the tutorials? Are they useful? Want more?

Please support me on Patreon!

Or make a direct donation!

made by Jasper Flick