Precision Surface Effects

You should look at the examples, especially about the CollisionSounds script. Because a lot of the default script values are arbitrary, but the proper values are assigned to the prefabs, depending on what sounds good, but that means that the values aren't magic numbers either, they're just the numbers that they were left at after solving the audio problems that came up.

There is one major thing you should keep in mind, the asset is designed to have the most control possible, which means there can be:

Many variables, including many multipliers. You can and should ignore them unless you want to control them.

Quite a complicated sampling system code-wise (But I've given some powerful scripts that do it for you). For example, this is the simplest way to play a sound:

```
var outputs = soundSet.data.GetRaycastSurfaceTypes(pos, dir);

outputs.Downshift(); //albeit this function doesn't affect the 1st output, so it's not necessary for this

soundSet.PlayOneShot(outputs[0], audioSource);
```

(For "pos" you can use `transform.position`. For dir you can use `-transform.up`, `Vector3.down`, or `Physics.gravity`)

And the reason it's so lengthy is because:

- There are many different things you could do with the information, so it is modular.
- I give the ability to get more than 1 output.
- DownShift is used to smoothly cull the list to the maxCount (default is 1) and using minimumWeight (default is 0). Until you do this, it is not guaranteed to be maxCount or fewer.

If you need more control than the (following) scripts, you should start getting comfortable knowing the parameters you have control over though. This is the equivalent of the last line (`soundSet.PlayOneShot(outputs[0], audioSource);`):

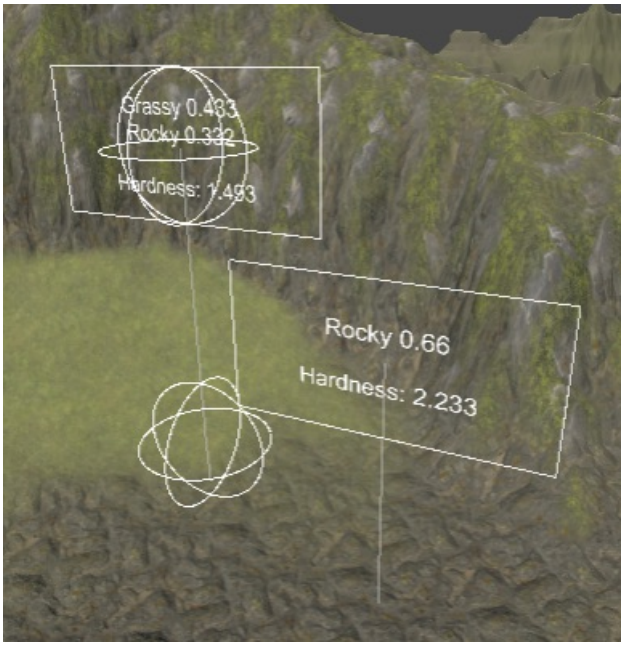
```
var output = outputs[0];

float volume = output.volume * output.weight;

soundSet.surfaceTypeSounds[output.surfaceTypeID].PlayOneShot(audioSource, volume, output.pitch);
```

The aforementioned scripts/components are:

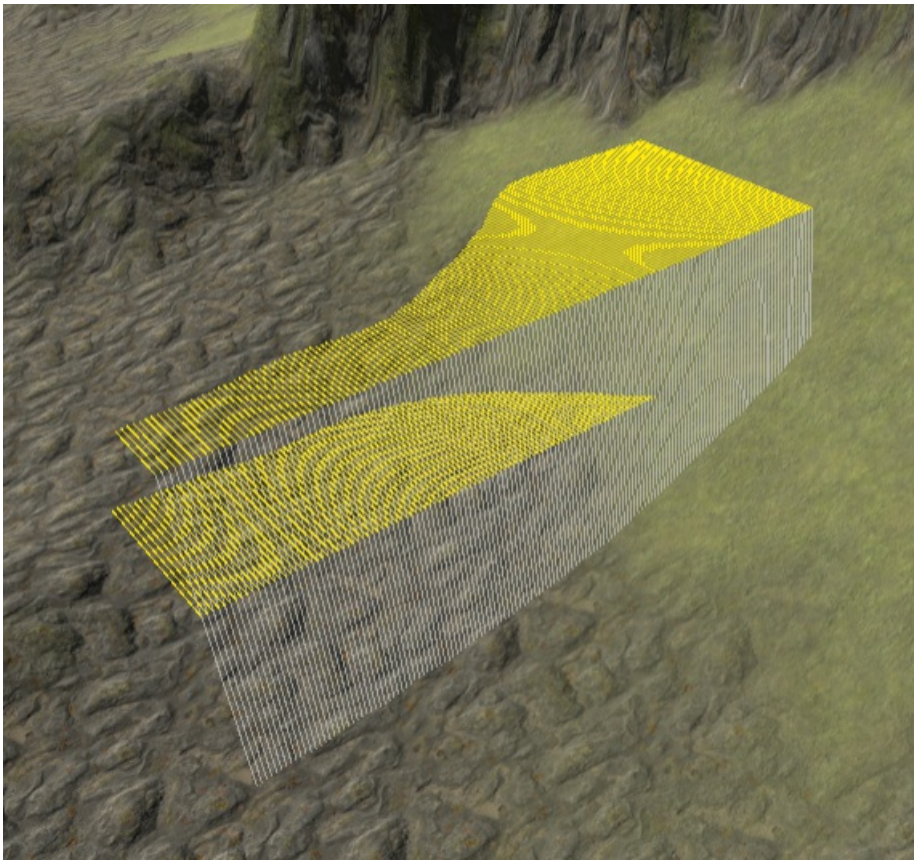
- **SurfaceEffectsBase** - This is a basic class you can inherit from, and then you'd call the function `Play(audioSources, pos, dir, impulse, speed)` and it would take care of Sounds and Particles using a Raycast.
- **GunShooter** is a good demonstration of implementing from **SurfaceEffectsBase**. The `Play` function returns the SurfaceOutputs it finds, and **GunShooter** uses that to relocate the AudioSources to the impact position. If you need your gunfire to be more rapid, you could cycle through a couple AudioSource sets, or perhaps you could use an AudioSource pool.
- **RaycastFeet** / **SpherecastFeet** - For footstep sounds. You can pair it with a customized version of the script **%Examples - Trash%/Example Scripts/SimpleFootsteps.cs** for even easier use. The reason I say to customize the script **SimpleFootsteps.cs**, is because it's currently designed around a single, specific, setup, which might not be how your setup is. Alternatively you could use the script alone, and use animation events to trigger the PlayFootSound(footID) function, however that wouldn't modulate the volume/pitch depending on the speed: walking/running/crouching, so instead you could use a setup like the script: **AnimatorEventFootsteps.cs**
- **CollisionSounds** - For impact and friction sounds/particles, as well as vibration sound.

And for testing/debugging purposes:

- **CastSoundTester** - For Ray/SphereCast testing. You should just use the **"Cast Sound Tester"** prefab.



- **ArraycastTester** - For testing of weight blending using a line of raycasts. You should just use the **"Arraycast Tester"** prefab. This script is not performant, so you shouldn't leave Arraycast Testers just laying around. They do disable themselves on start though.



- **CastSoundPerformanceTester** - Tests how long it takes to sample x number of times. Adding a marker to the object being tested should improve the speeds. You should use the **"Performance Tester"** prefab.

Weights and Blends

Depending on "maxOutputCount", you can receive more than 1 SoundType. The highest weighted sounds will be given priority when culling the list to the max count. You are also given control over the minimumWeight. Any sounds with lower weight than this are smoothly culled away, this means that even if a surface has e.g. 1% weight of dirt, it doesn't have to waste resources playing that sound (so long as the minimumWeight you set is 0.01 or larger).

"Blends" refers to a set of surfaceTypes which have individual control over their respective weights, which are internally normalized. For example you could have a Blends group of 2:

- "Grass", weight 2, normalized weight: 0.66666
- "Stone", weight 1, normalized weight: 0.33333

And what that means is that the sounds will have 0.66666 and 0.33333 as their volumeMultipliers. If you set maxOutputCount to 1, the weights are NOT renormalized. The volume will still be 0.66666 "Grass".

The string references are searched for keywords that belong to the SurfaceTypes, as usual.

It is not a good idea to use 50%/50% weighting if you plan to frequently use maxOutputCount=1

I would recommend using 2, maybe 3 maxOutputCount. For CollisionSounds.cs this count is set by the number of AudioSources you give "impactSounds" and "frictionSounds". You could provide the same "impactSounds" AudioSource multiple times (because it uses PlayOneShot), however the pitches will all be the same, which breaks the support for different Clip pitchMultipliers.

Gradiented Blending

Terrains have their own splat blending system with their alphaMaps, which works together with this Blends system; they are accumulated for each relevant splat, and then sorted and culled. The sampling used for Terrain's alphaMaps is bilinear, and you can see that in action using the Arraycast Tester prefab.

The component SurfaceBlendMapMarker allows similar functionality for non-convex MeshColliders, although I don't give support for generating the Texture/s, which should ideally be normalized and complete (the sum of all the BlendMaps' channels equaling 1).

Markers

Markers are used to override which SurfaceType/s a (non-Terrain) object represents.

- **SurfaceBlendOverrideMarker:** This overrides the Blend of specific submeshes (by ID). You can override whichever of the submeshes you want, you don't have to override them all. If you don't override a specific submesh material, then it will attempt other methods of finding the SurfaceType/s. This requires a non-convex MeshCollider, MeshFilter, and MeshRenderer. Be warned that if the material count/order changes, the indexes might no longer valid (and you'll need to reassign them).
- **SurfaceBlendMapMarker:** This uses one or more Texture's RGBA channels to blend between 4 SurfaceBlends (the container for blends), so long as the material ID is included in the blendMap. The channels are clamped/normalized above a sum of 1, so if the color sampled is (1,1,1,1) (white) then each of the BlendMap's SurfaceBlends (r, g, b, and a) will have 25% weight. You can create multiple BlendMaps if you need more than 4 blend types, and you can give them different relative weights, however, a material ID will need to exist in each of those BlendMaps for those weights to be relevant to the material ID.
- **SurfaceTypeMarker:** This overrides an entire object's SurfaceType in the simplest way, with no control over volume and pitch multipliers, and no blending, only one type.
- **SurfaceBlendMarker:** This overrides an entire object's SurfaceType Blends.

SurfaceTypeMarker/**SurfaceBlendMarker** will be fallbacks in the case of a non-convex MeshCollider. This is the attempt sequence (it will stop once it has gotten results):

1. **SurfaceBlendMapMarker**
2. **SurfaceBlendOverrideMarker**
3. Non-convex submesh materials
    - SurfaceData's materialBlendOverrides
    - Name searched for keywords
4. **SurfaceTypeMarker**/**SurfaceBlendMarker**
5. First material
    - SurfaceData's materialBlendOverrides
    - Name searched for keywords
6. Default SurfaceType

#1, #2, and #3 only apply to non-convex MeshColliders when using Raycast/Spherecast (Collisions don't know the triangleIndex so it can't find the submesh). Only #4 and #6 apply when the GameObject doesn't have a MeshRenderer.

SurfaceTypeMarker/**SurfaceBlendMarker** don't need a MeshRenderer. In the case that a collider doesn't have a MeshRenderer, it is advised to use them (unless you want to default to the default SurfaceType).

If you want to use CollisionSounds.cs with the support for non-convex MeshColliders disabled, then you can use **SurfaceTypeMarker** or **SurfaceBlendMarker** to provide the default SurfaceType/s, whereas footsteps (which use Ray/SphereCasts) will still be high quality.

Hardness

Hardness is a part of the SurfaceOutputs result that you get from sampling. You can use it to apply less damage on impact if you e.g. fall into dirt vs concrete. It also affects how big the particles of an impact are.

To me I think of the (arbitrary unit) of hardness to be similar to this:

- Flesh: on average .25 (although bones can make the hardness more like .5)
- Dirt: .5
- Wood: 1 - I generally use wood as the mental "reference" for 1
- Hollow Metal: 2
- Rock: 3
- Metal: 4

But again, it's a very arbitrary unit. For CollisionEffects' approximate impact duration calculation (which uses hardnesses), you have control over the constant multiplier: `public static float IMPACT_DURATION_CONSTANT = .1f;`.

Particles

SurfaceParticleSet Asset

This is an asset that defines the particles to be used in an interaction with certain SurfaceTypes

`SurfaceData data`

- This is the SurfaceData asset used to define the SurfaceTypes

`SurfaceTypeParticles[] surfaceTypeParticles`

- This is auto-resized to the data's surfaceTypes. You will have to reorder them yourself if you reorder the data's surfaceTypes
- These define some particle interaction settings for the given SurfaceType:
    - `Particles[] particles`
    - A class containing some settings for particles.
        - `SurfaceParticles particles`
            - The SurfaceParticles prefab to be used
        - `ParticleMultipliers selfMultipliers` - `ParticleMultipliers otherMultipliers`
            - The count/size multipliers for "self"/"other" (read the page "Self vs Other" if you don't know the difference)
        - `OriginType originType`
            - Other: if these particles are created from the other object, the collidee. Self: if these particles are coming from self, the collider. Both: if you want the particles to be applied to both the "self" and "other" perspectives; ie, if this SurfaceParticleSet is designed to be used for an interaction from "self" SurfaceType x, against the "other" SurfaceTypeParticles x, where x==x (metal-metal, wood-wood, plastic-plastic). Or more succinctly: if this SurfaceType is the same as the one this SurfaceParticleSet will be used on.

SurfaceParticleOverrides Asset

This is an asset that overrides the particles used for an interaction with given ParticleSets, against the objects that are relevant to the Blends that reference this (A (blend-using) Marker's gameObject, gameObjects that use a material used in a material blend override, etc.).

`Override[] overrides`

- Override is a class that overrides the interaction for a given SurfaceParticleSet:
    - `SurfaceParticleSet particleSet`
        - The ParticleSet that this override applies to
    - `Particles[] particles`
        - Same as before, but used for this override

SurfaceParticles Component

`SurfaceParticles[] children`

- You can call children SurfaceParticles prefabs, so that you can for example mix in some dusty particles along with fragment particles. A children SurfaceParticles doesn't call it's own children (this is essentially just to prevent the possibility of infinite loops.)

`bool inheritVelocities`

- This should always be enabled unless you have performance problems

`float inheritAmount`

- This is how much the velocity inheritance diverges from the (weighted) average velocity, that means if it is 0, then all the particles inherit precisely the weighted average velocity, if it is 1 then particles inherit randomly between the object A and B's velocities. "Weighted average velocity" is the velocity if the inertia of both objects are averaged; if a ball hits a static collider then the weighted average velocity is 0, because the mass of the kinematic collider is "infinity".

`Vector2 inheritSpreadRange`

- This is similar to `inheritAmount`, but it gives the ability to shift the inherited velocity toward one of the objects, such as if you set the range to be (0.5, 1), and it doesn't care about the weighted average velocity: if you set the range to be (0.5, 0.5) then the velocity inherited will always be the straight up average between the velocities.

`bool setColor`

- Whether or not you want the color to be multiplied into the particles. This shouldn't be the case if the particles represent for example dust clouds or sparks. This actually multiplies the startColor/s of the ParticleSystem (supports the modes "Color" and "Random Between Two Colors")

`float shapeRadiusScaler`

- Collisions find an approximated collision surface radius (used in the ParticleSystem's shape module) by finding the average distance of the contact points to the average contact position. This is not a very good model, because e.g. boxes don't have a circular contact area. This field multiplies the radius found; it should perhaps be something like `0.5`.

`float constantShapeRadius`

- This number is added to the radius. It doesn't matter how many contacts or how large of an area the contacts cover, this number will always be added just the same. Perhaps this can be used to give a sort of exterior shockwave effect.

`Vector3 shapeRotationOffset`

- This rotation is an offset to the ParticleSystem's shape module.

ColliderEffects' Speed Fading

`float impactSpeedMultiplier` - `float rollingSpeedMultiplier` - `float slidingSpeedMultiplier`

- These multiply the speeds used for CollisionEffects' Particles' Speed Fading range. The reason for this control is that: sparks need sliding friction to be created (rollingSpeedMultiplier ~= 0), whereas dust doesn't care as much about whether the speed is from sliding or rolling. Just a couple more controls to get control over undesired behaviour.

Speed

`float constantSpeedMultiplier`

- This is a constant multiplier of the ParticleSystem's main module's start speed. This is useful if you want particles to always be created with a minimum speed even if the CollisionEffects' object is moving very slowly

`float speedMultiplierBySpeed`

- This is how much the collision speed affects the start speed multiplier (this is the code used `mainModule.startSpeedMultiplier = startSpeedMultiplier * (constantSpeedMultiplier + speed * speedMultiplierBySpeed)` where `startSpeedMultiplier` is cached at Awake).

Count

`ScaledAnimationCurve rateByForce`

- The class `ScaledAnimationCurve` is essentially a lightweight helper class to deal with AnimationCurves easier. This is the code used:

```
public float Evaluate(float t)

{

    return constant + curveMultiplier * curve.Evaluate(t / curveRange);

}
```

- `rateByForce` is how many particles should be emitted per second while force t is applied
- You need to be careful with the points of the AnimationCurve. The curve would like to be evaluated from the range (0, maxForce) where maxForce is the largest forces you want. The easiest thing is to set `curveRange` to be maxForce, be it something like 10000, and set the AnimationCurve to be one of the built-in/default curves within the x range of (0, 1) and the y range of (0, 1). Your job is to shape the curve how you want, how convex, how concave, how whatever, but it would be good to remain within the x(0, 1) y(0, 1) range, and let the other more readable fields do the offset (constant) and scaling.

`float countByInverseScaleExponent`

- Further down you will see controls to scale the particles by the forces involved. This is used to decrease the number of particles depending on how large the particles are. The number of particles emitted is multiplied by: `1 / Mathf.Pow(scale, countByInverseScaleExponent)` where `scale` is the size multiplier for the particles.

`ScaledAnimationCurve scalerByForce`

- This is used to make the particles larger when the forces are larger

Self vs Other

Blends, Collision Effects, Particle Overrides, and Particle Sets all have control over a multiplier struct called `ParticleMultipliers`

- `float countMultiplier`
- `float sizeMultiplier`

However they are separated into 2 fields:

- `ParticleMultipliers selfParticleMultipliers`
- `ParticleMultipliers otherParticleMultipliers`

No matter where you find these fields, they all mean the same thing:

- "Self" -> The object querying/The colliding object
- "Other" -> The object being queried/The object being collided against

The difference is: **Self is the object that is sampling the surface types of the other object.**

- Collision Effects:
  - If A has a CollisionEffects component and B doesn't, then A is "self"
  - If both have CollisionEffects, then:
    - If A has a higher priority, then A is "self"; if B has a higher priority, then B is "self"
    - If they have the same priority, then it is "random" whether A or B will be "self" within a frame (and the other object being "other")
- Footsteps
  - The object creating footsteps is "self"
- GunShooter
  - GunShooter is "self"

You *need* to remember that "self" means the object that is testing for SurfaceTypes in the object "other", because you have the 2 multiplier fields in most of the **Markers**, and intuitively "self" feels like it should represent the object the Marker component is attached to, but it does not. ***"Other" is always the object being queried. "Self" is the object that is querying.***

The object "other" needs to indicate what SurfaceType it represents; it needs a Marker or a valid Material name.

Flipping

In ParticleSets and ParticleOverrides, you have control over the enum OriginType The idea is that a SurfaceParticles prefab is designed around either:

- The damage particles coming from "other".
- The damage particles coming from "self"
- Both

And by "damage" particles I mean shrapnel/fragments/smoke/feathers/sparks/etc.

The perspective of how you make these prefabs matters, sparks are particles created FROM a metal ball striking stone. Stone particles are created FROM the stone being striked.

When designing the prefab, the perspective is at the default: Other. You can easily just change the perspective using the `rotationOffset`

Giving control over the perspective allows you to properly use assymetrical ParticleSystem shapes if you want. It also means the size and count multipliers are applied only to the particles you want.

Collision Effects

Warning: Using CollisionEffects on a fast downward rigidbody which is not 'Continuous Speculative', (and especially if the impactCooldown is higher than 0), can make the wrong sounds if a collider it will hit is too thin and close above the (for example) TerrainCollider. This happens because the discrete physics steps happen too infrequently to resolve the Collider before it reaches the TerrainCollider below (so it makes the sounds expected for colliding with the TerrainCollider, and if the cooldown is higher than 0 it can additionally prevent the proper Collider's sound from playing).

Speculative impacts must be enabled for 'Continuous Speculative' Rigidbodies.

`doImpactByForceChange` is an inferior alternative to `doSpeculativeImpacts`. You shouldn't need both enabled.

The **maxVolume** settings are essentially only to make the game more pleasant, because with massive objects the volume would hurt your ear, especially so in real life with such huge impacts, but games are meant to be fun.

If a lot of sounds are occurring, there can be quite a lot of popping if you don't increase your "Max Real Voices" in project settings' audio tab

CollisionEffects Script

`bool findMeshColliderSubmesh`

- If enabled, it will use a raycast to find the submesh of a non-convex MeshCollider

Impacts

`float impactCooldown`

- This prevents multiple impacts shortly after each other

`bool doSpeculativeImpacts`

- This predicts when a new impact should occur during OnCollisionStay. This is useful because frequently an object will tilt and fall while remaining in contact, but a sound should still be made. `float separationThresholdMultiplier`
- The default bias threshold used to determine if a ContactPoint is in contact or not is 0.01 meters. Depending on the scale of your objects you might need to increase or decrease this multiplier. `float minimumAngleDifference`
- The minimum angle a new ContactPoint has to be from all the other ContactPoints, to be considered a new impact.

`bool doImpactByForceChange` - `float forceChangeToImpact`

- This is an inferior alternative to `doSpeculativeImpacts`. This allows an impact to be generated during OnCollisionStay if the change in forces is sudden enough.
- This is that threshold, the change required

Sounds

`SurfaceSoundSet soundSet`

- This is the soundSet used for the Impact and Friction sound

Scaling

`SoundScaling soundScaling`

- `float soundSpeedMultiplier`
- `float soundForceMultiplier`
- `float totalVolumeMultiplier`
- `float totalPitchMultiplier`
    - These are used to easily scale certain aspects of the sounds, for example if you want to scale up a sphere you likely want to:
        - Scale the pitch down
        - Scale the speed down
        - Scale the force down
    - Volume multiplier is almost the same as force multiplier, however the force multiplier also affects the force tested against `forceChangeToImpact`.
    - These scalers apply only to sounds, not particles.
    - These scalers do not apply to the vibration sound, but you can synchronize the multipliers with a CEVibrationScaling.cs component (CE for Collision Effects) (That script only applies in editor mode however, not runtime). The reason is that a vibration sound should be able to receive forces from sources other than the CollisionEffects, such as bullet impacts.

Impact Sound

`AudioSource[] audioSources`

- You specify which audiosources you want to play sounds with. They can be all the same because it uses PlayOneShot, but that would break support for different pitchMultipliers. `float minimumTypeWeight`
- The minimum SurfaceType weight to consider

Volume

`float volumeByForce`

- How much force (multiplied by forceMultiplier btw) for how much volume

`Vector2 volumeFaderBySpeedRange`

- The range of speed that fades the sounds. Any speed at or lower than the `x` value multiplies the volume by 0, any speed higher or equal to the `y` value multiplies the volume by 1.

`float maxVolume`

- To clamp the volume, to prevent massive objects from irritating the player's ears (like they would realistically, in real life).

Pitch

`float basePitch` The default pitch at 0 speed

`float pitchBySpeed` This increases/decreases the pitch by the speed. (I would recommend decreasing for impact sounds)

Friction Sound

`bool doFrictionSound`

- You can disable friction sounds (for performance)

This shares a lot of similar fields to the Impact Sounds, these are the differences: `AudioSource[] audioSources`

- They can't be all the same

`float pitchBySpeed`

- It should only be a positive value for friction sounds.
- Be careful with this, because it can make the sounds feel robotic if the value is too high.

`float maxForce`

- This clamps the forces so that it doesn't get hung up on too large volumes for too long.

Amounts

`float rollingAmount` - `float slidingAmount`

- These multiply the rolling speed(velocity of the Rigidbody's center)/sliding speed(velocity at the contact point) respectively (afterwhich they are added together). This is then used with volumeFaderBySpeedRange to fade the volume of the friction sound.

Rates

`float clipChangeSmoothTime`

- This is the time in seconds that it takes to fade away a friction sound when the AudioSource is wanted for another SurfaceType

`SmoothTimes volumeSmoothTimes`

- This contains `float up` and `float down`, which are the time in seconds that the volume is smoothly interpolated, up if the target volume is higher than the current volume, down if not. I believe it is good to separate up/down control. This is especially important because the physics (FixedUpdate) interval is not the same as the default, Update.

`float pitchSmoothTime`

- This is the time in seconds to smoothly interpolate the pitch

`float frictionNormalForceMultiplier` This multiplies the force that is along the contact normal, because friction sound generally cares most about forces perpendicular to the surface normal.

Vibration Sound

`AudioSource audioSource`

- The AudioSource used for the vibration sound

`bool doVibrationSound`

- You can disable the vibration sound (for performance)

`VibrationContribution vibrationContribution`

- These allow you control over how much of Friction/Impact is taken into account
    - `float frictionForceMultiplier`
    - `float frictionSpeedMultiplier`
    - `float impactForceMultiplier`
    - `float impactSpeedMultiplier`

- Friction Sound only contributes if it's enabled

Particles

`ParticlesType particlesType`

- Enum that allows you to decide when to create particles within `None, ImpactOnly, ImpactAndFriction`

`SurfaceParticleSet particleSet`

- Which ParticleSet to use for particles. If the SurfaceData assigned to it is not the same as the one assigned to the SurfaceSoundSet `soundSet`, it will sample the SurfaceTypes twice, which is not very performant.

`float minimumTypeWeight`

- The minimum SurfaceType weight to consider

`float selfHardness`

- Different surfaces can define the hardnesses they represent. (I would have liked to do a (proper) physically based system for PSE (but PhysX is not good enough for it), and this is a sign of my wishes for it). An impact's duration is not the same as Time.fixedDeltaTime, nor should it be 0 like PhysX attempts to resolve things to, but instead it (should) take place over a longer period of time. For example, a metal sphere hitting dirt should take a longer time to bounce than if it hit concrete. So to find my (extremely) approximated impact duration, I use this: `IMPACT_DURATION_CONSTANT / Mathf.Max(0.00000001f, particles.selfHardness * soundOutputs.hardness);`. The reason to find the impact duration is to find the peak forces involved (force = impulse / duration) to scale the particles in the SurfaceParticles script, so that bigger collisions (bigger forces) should have bigger particles. `Color selfColor`
- This is used to color the particles if the particles are considered "self" particles

`ParticleMultipliers selfMultipliers`

- These multiply the count and size of particles that are considered "self" particles

`ParticleMultipliers otherMultipliers`

- These multiply the count and size of particles that are considered "other" particles

`float minimumParticleShapeRadius`

- This is added to the radius calculated for the ParticleSystem's Shape, so if the calculated radius is 3 and this is .3, the total radius of the shape is 3.3. The reason for that is a big sphere would likely have a larger contact area with the ground (because of deformation, which is not a concept in PhysX), and therefore the particles should be created around a larger area.

`Vector2 faderBySpeedRange`

- This fades the number of particles created depending on the speed. (the multiplier `soundSpeedMultiplier` doesn't affect it)

Vibration Sound

These can be applied to static geometry.

You should always use a **VibrationMarker** to point to a vibration sound, otherwise if something collides it won't know if or where a VibrationSound exists. You should add the component even if you are using a **CollisionEffects**, because a higher priority **CollisionEffects** will need to know the existence of the other object's vibration sound.

BasicVibrationSound.cs

`SoundScaling scaling`

- Scales the speeds and forces given. It is probably a good idea to use the script CEVibrationScaling.cs (Collision-Effects Vibration Scaling) to synchronize the multipliers from this component and the CollisionEffects component.

`VibrationContribution contribution`

- `float frictionForceMultiplier` - `float frictionSpeedMultiplier` - `float impactForceMultiplier` - `float impactSpeedMultiplier`
  - These allow you control over how much of Friction/Impact is taken into account

`float volumeByForce`

- Scales the volume by the forces involved

`float basePitch`

- The default pitch at 0 speed

`float pitchBySpeed`

- Increases the pitch by the speeds involved

`float smoothTime`

- This is used to interpolate the pitch and volume. The reason it's needed is because the physics (FixedUpdate) interval is not the same as the default, Update. This is mostly put into effect for when the volume increases (such as on impact) because the decay rates smoothly decrease the volume.

`float maxForce`

- This is used to clamp the forces stored, this can be important if let's say an object is hit with a gajillion newtons, the decaying would take a long time to fade the volume, because although the volume would like to be ear splitting (let's say 100000 volume), it would have to be clamped to 1 or lower, so it would ring at the same volume for a very long time.

`float maxVolume`

- To clamp the volume, to prevent massive objects from irritating the player's ears (like they would realistically, in real life).

(Force) Decay

`float exponentialDecay`

- The decay that doesn't regard itself with the size of the current force, it is a multiplier like this: `*= 1 / (1 + Time.deltaTime * exponentialDecay)`

`float linearDecay`

- Linearly decreases the force. This is helpful to silence the volume quickly, because the exponential decay would never reach 0, and at small values it would remain an active AudioSource (although I do check audibility based on the arbitrary constant: `0.00001`).

Footsteps

RaycastAnimatorFeet:

- You should probably override the foot direction, because if the sampling timing is not properly synchronized, it might sample while the feet are pointed almost horizontally.

Coding Yourself

A lot of the code is not designed for updating the variables with scripts at runtime, for example there is some caching for performance going on, but I've tried making some components use a Refresh() method that you can call to recalculate things at runtime if you've changed the fields. I feel like most of the code has no need to be updated at runtime, but if there is an inability to do something you want because of that restriction, I can add more runtime accessibility.

Since Collisions don't know anything about the triangle index, unlike RaycastHits, for CollisionEffects.cs I allow the ability to automatically use a raycast to find it in the case that it would be useful to do so (non-convex MeshCollider). If you want to find the submesh SurfaceType/s yourself using that code, you can use the static method `CollisionEffects.GetSmartCollisionSurfaceTypes(Collision c, bool findMeshColliderSubmesh, SurfaceData data);`. This raycast distinction code is done with the first ContactPoint only, but that should be okay.