Animating a concert's crowd with hundreds of individuals using DOTS

Jérémy L. Auclair
jeremy.loic.auclair@gmail.com
CNAM-ENJMIN
Angouleme, France

ABSTRACT

In this article I will present how I simulated and rendered a huge crowd of concert attendees in Unity Engine while maintaining a stable and high framerate.

Key-Words: Unity Engine, C#, DOTS, ECS, Simulation, Crow, Animation

1 INTRODUCTION

Unity DOTS is shorthand for Data-Oriented Technology Stack, it is a set of tools given to the developers that uses Unity to get better performances.

1.1 Managed vs. Unmanaged

Before we explain any tool contained in DOTS, it is crucial to understand the difference between Managed and Unmanaged data.

C# has a garbage collector, meaning that once the data on the stack isn't used anymore, it will be recycled. Any types that are managed by the garbage collector are called managed types, and variables of these types will be managed variables, and types that aren't managed are unmanaged types and will create unmanaged variables.

Arrays, List, and classes are managed. Most of the primitive types and structs that only contain variables of managed types are unmanaged.

This is important because a lot of DOTS' tools' features will react differently and offer different possibilities depending on whether the data you are using is managed or unmanaged.

1.2 Unity Burst

Burst is a special compiler that will transform standard C# code into Native Code. Any function or class can be marked for Burst Compilation using a simple attribute, but if the function uses managed variables or types, Burst will not compile it.

1.3 C# Jobs System

By default, all the MonoBehaviour scripts a developer using Unity writes run on the main thread. The C# jobs system allows developers to run tasks on other threads, using the C++ Jobs System that the backend of Unity uses.

1.4 Unity ECS

The most complex of DOTS tools. ECS completely changes the paradigm of Unity. Instead of having GameObjects with MonoBehaviour attached to it that each runs their own code, ECS uses Entites, Compononents and Systems.
Entities has several components attached to it. Those components contain variables. And then the system queries the components and processes their data.
Entities exist in their own space separated from GameObjects. GameObjects will be loaded in a scene, and Entities will be loaded in their own sub scene.

In editor, Entities are showed as GameObjects and Components are added to them as MonoBehaviours scripts called "Authoring" scripts. These authoring scripts behave like regular MonoBehaviours, allowing developers to use the same techniques to create tools, gizmos etc. An authoring script is turned into a Component using a Baker, the Baker can be fully customized.

1.5 Combining tools

Those tools aren't separate, you can all use them independently from one another, but they reveal their full potential when you use them in combination. You can, for example, query all components of a given type, process their data in parallel using the Jobs System, and compile the Jobs using Burst for maximal performance.

2 ANIMATION

The first thing we want to do with our crowd of NPCs is animate them, they need to jump up and down. Unity comes with its packaged animation system: the Mecanim Animation System, it doesn't use any DOTS technology. If we try to spawn 1000 NPCs and animate them using Mecanim, we realize that the framerate is very low. We need to use another solution, but which?

For a time, Unity offered its own DOTS animation animation package, but its support was dropped with ECS 0.50 at the beginning of 2022. An ECS-based Animation system is on the roadmap of DOTS, but no date is given.

There are several options, but we'll use Latios Framework's Kinemation. Latios Framework is a package made by a third-party developer that is open source and regularly maintained. It offers various tools usable with ECS, but we'll only use Kinemation.

2.1 The basis of animation

Animation in 3D engines works by having a mesh and a skeleton. The skeleton is a group of bones that can be rotated, and in some cases moved or scaled. Each vertex of the mesh is bound to multiple bones and will be moved along with those bones. The impacts each bones have on a vertex is defined by the bone's weight relative to the vertex.

2.2 Latios Framework's Kinemation

Kinemation bakes the information of the mesh to animate: the animation clip to use, the skeleton etc. And moves the vertex using a custom node in Shader Graph.
Two custom nodes can be used:

- Vertex Skinning: Only usable with models where the skin weight (how many bones affects any given vertex) is 4, the standard for Unity.
- Deform Skinning: Usable with values of skin weight ranging from 1 to 255 (inclusive)

All the materials used by the mesh need to use a shader graph where one of these custom nodes is plugged to the vertex shader part.

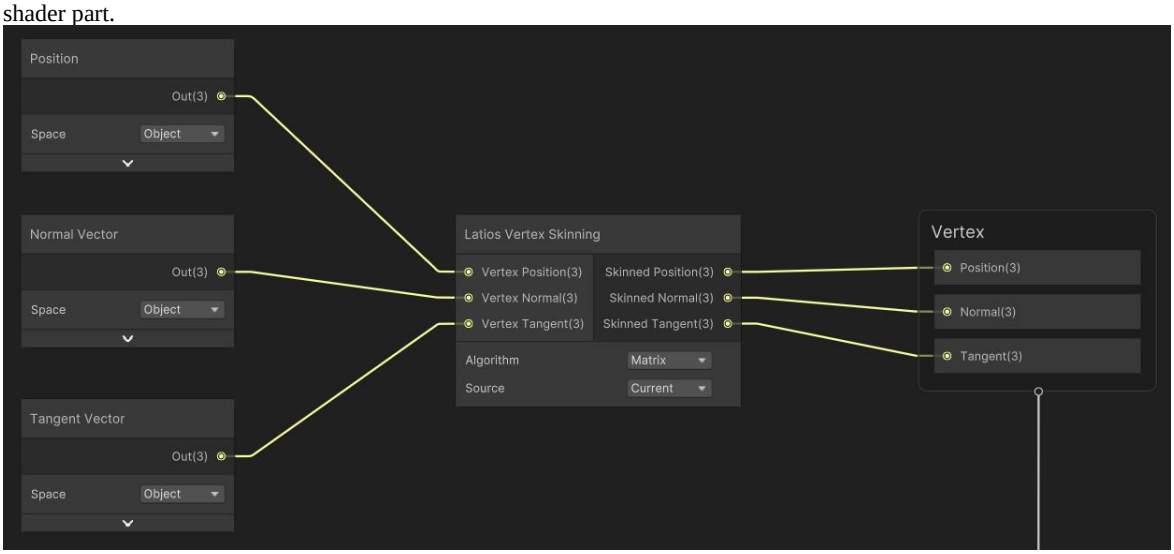

Figure 1: The custom node plugged to the Vertex Shader

Any animation clips used is then turned into BlobAsset (a variable filled with binary data readable at runtime by ECS) by a custom Baker.

Finally, a System needs to query all the bones of all the animated mesh and call the SampleBone function that takes a clip, a bone and a time relative to the start of the clip and moves the bones accordingly.

Snippet 1: Sampling bones; elapsedTime is the time elapsed since the application was launched

```
clipTime = (roSingleClip.Offset * clip.duration)+ (elapsedTime * roSingleClip.SpeedMultiplier);
clipTime = clipTime % clip.duration;
transform.localTransformQvvs = clip.SampleBone(boneIndex.index, clipTime);
```

I benchmarked three method of animation using various crowd sizes: regular Mecanim, Kinemation with Vertex Skinning and Kinemation with Deform Skinning.

The three methods were compared on the framerate of the application and the ram it allocated.

The application was built and then ran on a laptop computer running Windows 10 64 bit and equipped with 16 GB of RAM, an NVIDIA GeForce GTX 1660 Ti with 6 GB of VRAM and an Intel Core i7-9750H CPU.

On each test, the entirety of the crowd would be in camera range.

Between each test, the scene would be reloaded and the entirety of the systems of ECS reset.

At first, Kinemation Deform Skinning showed the same results as Kinemation Vertex Skinning. But once the 20 000 NPCs bar was reached, mesh animated using Kinemation Deform Skinning would not render at all, or render with sever visual glitches. As such, it was removed from the results.

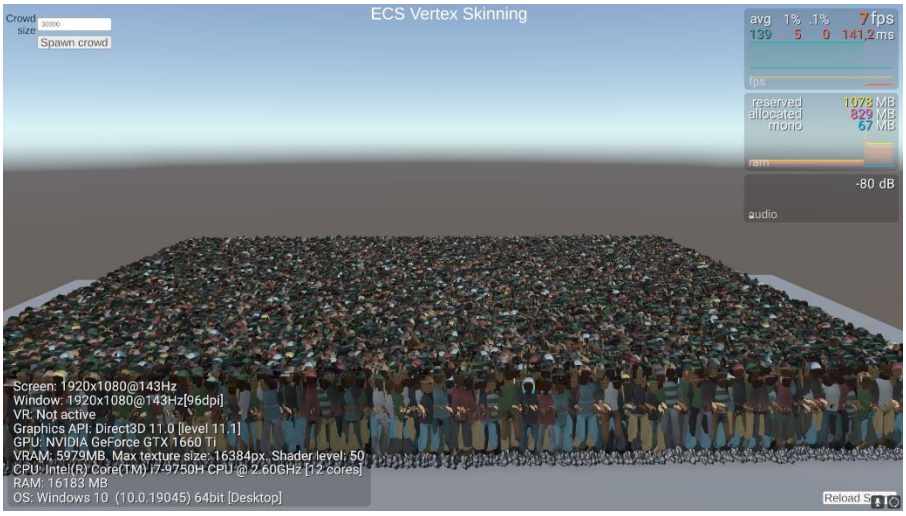

Figure 2: The crowd rendered using Vertex Skinning. No visual glitch is seen.

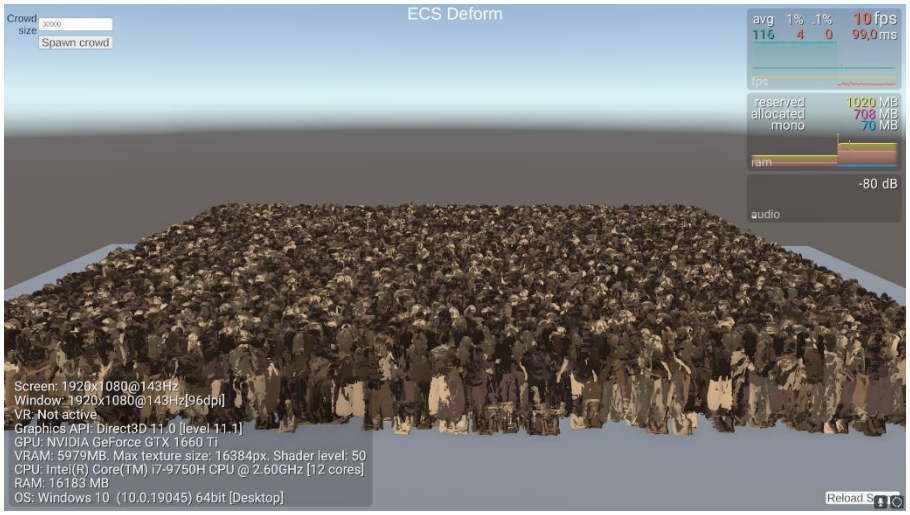

Figure 3: The crowd rendered using Deform Skinning. A sever visual glitch is seen

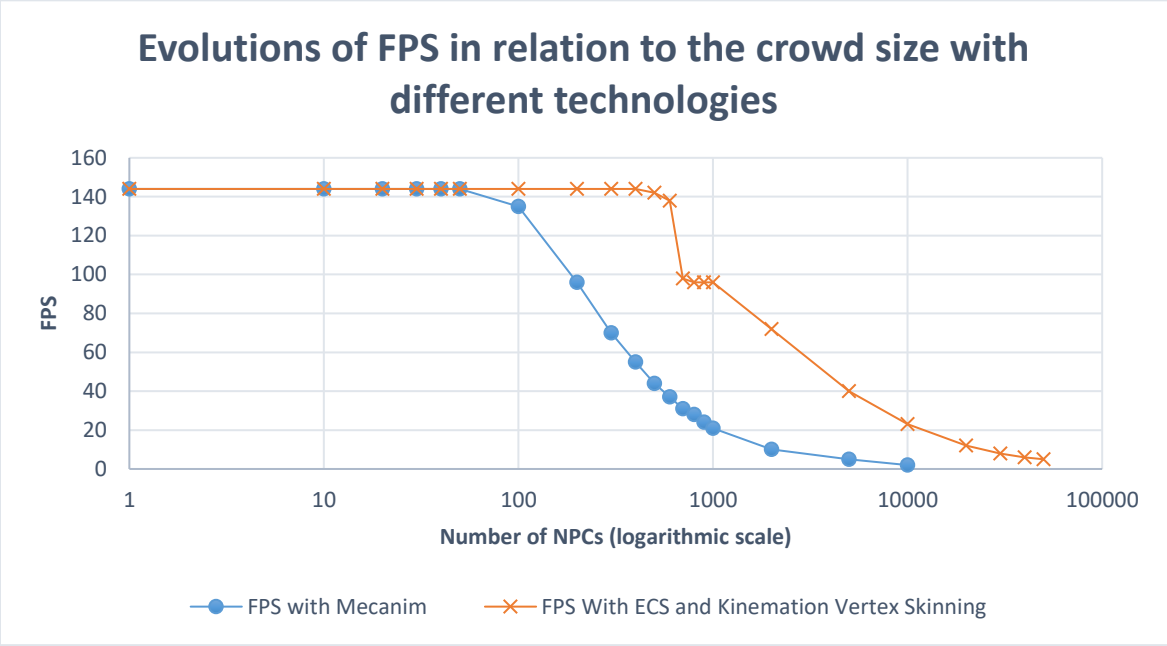

Figure 4: Evolution of FPS in relation to different technologies

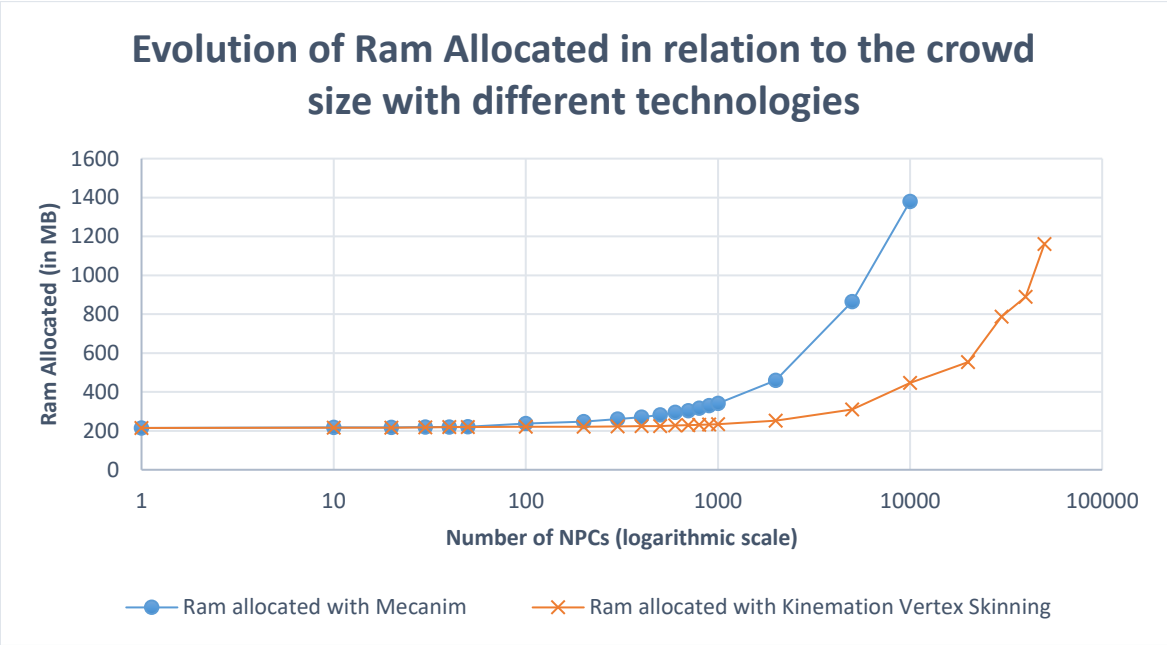

Figure 5: Evolution of Ram Allocation in relation to the crowd size with different technologies

As you can see, Kinemation with Vertex Skinning offers much better performance. While Mecanim can only handle 400 NPCs before dropping below 60 FPS (considered the expected amount of FPS by many gamers), Kinemation can render 2000 NPCs at 72 FPS.

2.3 Randomizing the animation to add variation

We now have a crowd of NPCs jumping up and down, except they are all synchronized, and all jumps up and down at the same time, it appears unnatural and unrealistic.

A solution is to randomly apply an offset and a multiplier to the animation time of each NPC.

Each NPC has a component containing the animation clip it must plays, “SingleClip”,all we have to do is to add an offset and a multiplier variable to this component . These values are to be randomized only once. In MonoBehaviour, we could simply randomize those values in the Start function, but here we cannot do that.

How a system works is that you query a list of components, and ECS will return you all the entities with the requested components, then you can do what you need to do with them.

So what we need to do is to add a second component, “SingleClipRandomConstraints” that contains the constraints of the randomization, we run a system that queries both the SingleClipRandomConstraints and the SingleClip, it sets the values of the offset and multiplier, we then remove the SingleClipRandomConstraints component from the NPC’s entity and this way, the system will not touch to this NPC again since it will not come up in the query.

Except that removing Components is a costly task, so instead we use IEnableableComponent

Snippet 2: SingleClipRandomConstraints

```
public struct SingleClipRandomConstraints : IComponentData, IEnableableComponent
{
   public float MinSpeedMultiplier;
   public float MaxSpeedMultiplier;

   public float MinOffset;
   public float MaxOffset;
}
```

Snippet 3: AnimationRandomizingSystem

```
public struct SingleClipRandomConstraints : IComponentData, IEnableableComponent
{
   public void OnUpdate(ref SystemState state)
{
   foreach ((var singleclip, var singleClipRandomConstraints, Entity entity) in
SystemAPI.Query<RefRW<SingleClip>, RefRO<SingleClipRandomConstraints>>().WithEntityAccess())
   {
```

```
    singleclip.ValueRW.Offset = Random.Range(singleClipRandomConstraints.ValueRO.MinOffset,
singleClipRandomConstraints.ValueRO.MaxOffset);
    singleclip.ValueRW.SpeedMultiplier =
Random.Range(singleClipRandomConstraints.ValueRO.MinSpeedMultiplier,
singleClipRandomConstraints.ValueRO.MaxSpeedMultiplier);
    singleclip.ValueRW.HasBeenRandomized = true;
    state.EntityManager.SetComponentEnabled<SingleClipRandomConstraints>(entity, false);
  }
}
```

This way, we can just disable the component instead of deleting it. It serves the same purpose, as it will not come up in query, but it takes a lot less processing time.

Now that we have a random offset and multiplier, we can just apply it to the time we use to sample the bone

Snippet 4: Sampling bones with random variations; elapsedTime is the time elapsed since the application was launched

```
if (roSingleClip.HasBeenRandomized){
  clipTime = (roSingleClip.Offset * clip.duration)+ (et * roSingleClip.SpeedMultiplier);
  clipTime = clipTime % clip.duration;
}
else
  clipTime= clip.LoopToClipTime(et);

transform.localTransformQvvs = clip.SampleBone(boneIndex.index, clipTime);
```

3 MIX AND MATCHING

Now all the NPCs are jumping at their own pace, but they still all look the same. So we make it that the basic NPC prefab has all the versions of a hat possible, all the versions of a jacket possible etc. And multiple materials for each.

We re-use the technic used on animation variation to run code only once on each NPC, we roll for a material and a piece of clothing per category, and we change the materials accordingly and disable the entities accordingly.

But we face two problems.

3.1 Managed types

Materials are managed types, so they can't be contained in regular components. Regular components can only store unmanaged variables. We could find a way to convert the materials to unmanaged variables, but we have an easier and sturdier solution: managed components. Managed components should be used scarcely, as they cannot be Burst compiled, and are overall more costly. But since this mix and matching component is only used at the start of the application once before being disabled, we can afford it and this way we're sure no memory leaks happen.

3.2 Deleting a lot of entities

We delete a lot of entities in just an iteration of our system: a dozen unused clothing item, multiplied by the number of NPC. ECS is built on arrays, deleting an entity modify those arrays, and that's costly. So, we can make it easier on the CPU by creating what is called an EntityCommandBuffer.

Instead of deleting an entity, we add a delete command on the buffer. And once the system has iterated on every NPC, we "Playback" this EntityCommandBuffer and everything is deleted in one call.

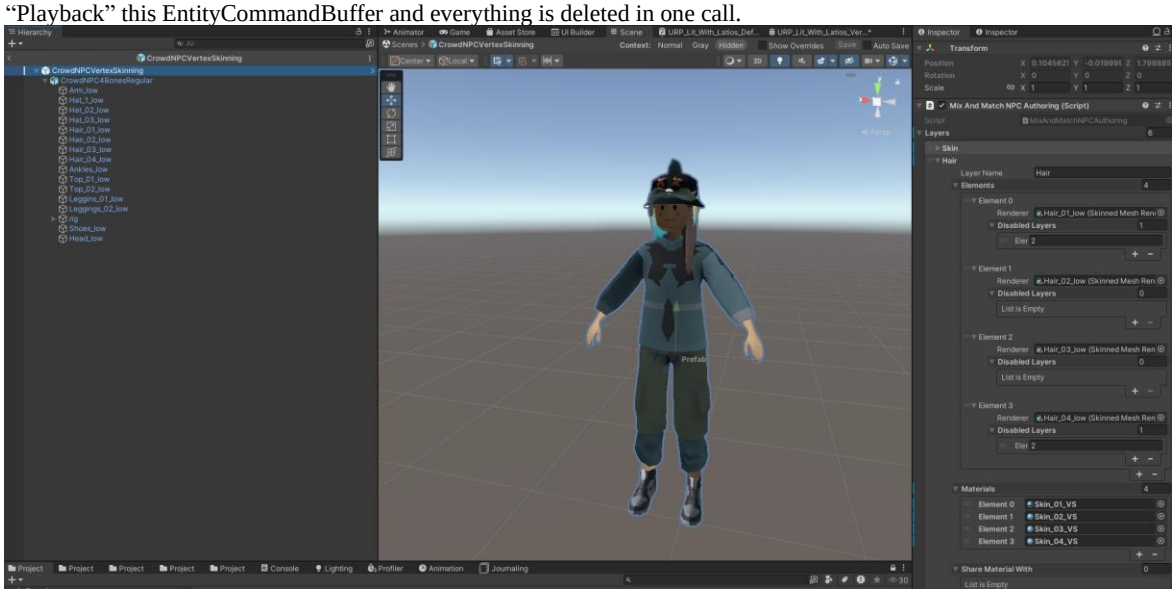

Figure 6: Crowd's NPC with all the clothing items on

4 PHYSIC SIMULATION

Now that we have our various NPCs with unique looks and unique timing to their jumps, we need to get them to push off each other in a chaos state, like in a real concert's mosh pit.

Since people cannot fly, we just need to simulate a 2D physics system.

4.1 Physics engine

First, we need to find a way to detect collisions. ECS comes with its own physic system, but it only supports 3D and there is no native feature to remove an axis from the simulation.

We only need to handle two kinds of collisions: circle to circle, and circle to 1-sided line (the "walls" of the simulation), so I opted for a simple homemade system. Every NPC is represented by a circle. They all share the same radius.

4.2 Parallelism and how to detect collisions

Until now, I mostly explained the System's behavior as iterating through Entities and treating each one after the other. But now, we need to treat entities on multiple threads at the same time. A frame of the physic simulation can take a lot of processing power. **To allow for parallel execution, we use the C# Jobs System.**

The C# Jobs System has an Execute function with Components as its parameters. When the job is scheduled, each thread will find entities with these Components attached and execute the code of the Execute function on them. Just like

queries, but parallel. In addition to the components, the Job has access to variables that the System sends to it. We will call each execution of the Execute function on an entity a “Job Instance” for simplicity’s sake.

That is how we will detect collision, the system will send to the Job the radius of all the NPCs as well as an array of their position. Then each Job Instance will compare the positions with the NPC it is treating and if the distance is too low, an NPC-to-NPC collision occurs. If the NPC oversteps the boundaries of the walls, its direction will be flipped to keep it in.

There is however a problem with that: since threads runs in parallel, the array each Job Instance look is often not accurate anymore, this is an intrinsic flaw that I don’t see how to correct.

4.3 How to react to collisions

In a mosh pit, people bounce off each other with create waves and propagates motion through the whole crowd. Like balls in a game of pool. I based the collisions reactions off of a pool simulation created by Youtube Channel Ten Minute Physics.

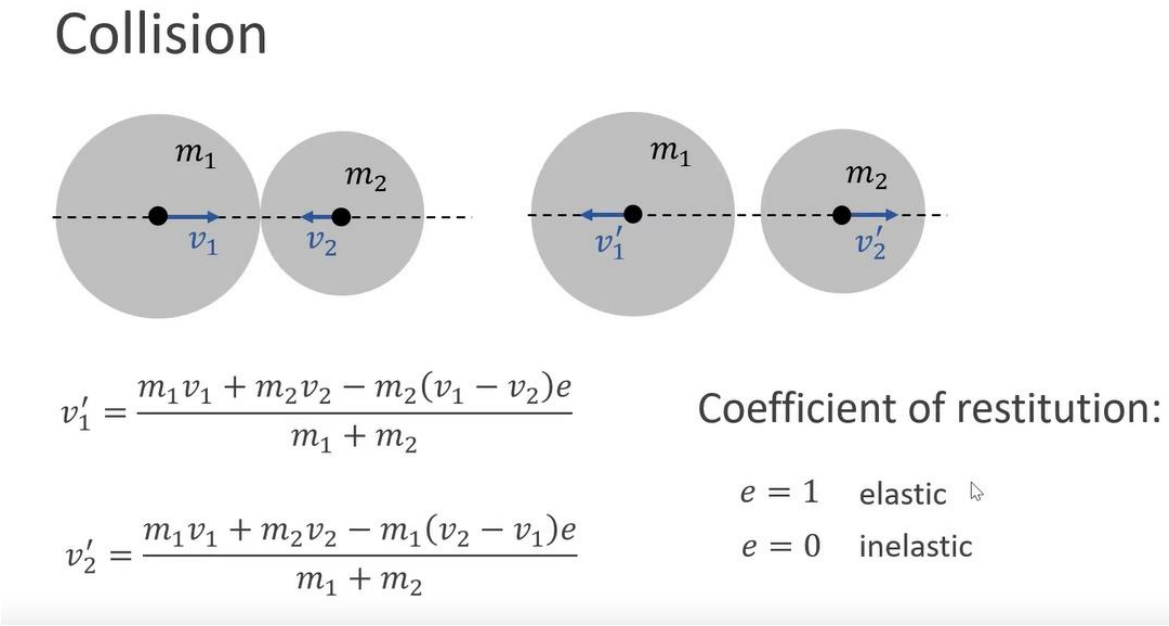

Figure 7: Collision equation from the video "03 - Writing a billiard simulation in 10 minutes" (https://youtu.be/ThhdlMbGT5g)

4.4 Randomizing the physical properties of the individuals

There are however differences between a human in a mosh pit and a pool ball. Humans have a distance they like to keep between them and the scene, they try to dampen their movement, and their weight is different from one another.

Now, people will dampen their velocity, except the part of their velocity that brings them closer to the distance they prefer to the stage.

All these new variables are randomized but linked, instead of all being rolled independently, a value between 0 and 1 is rolled, and used as the X axis on a random curve for one value, then it is slightly altered and used on another curve for another value etc.

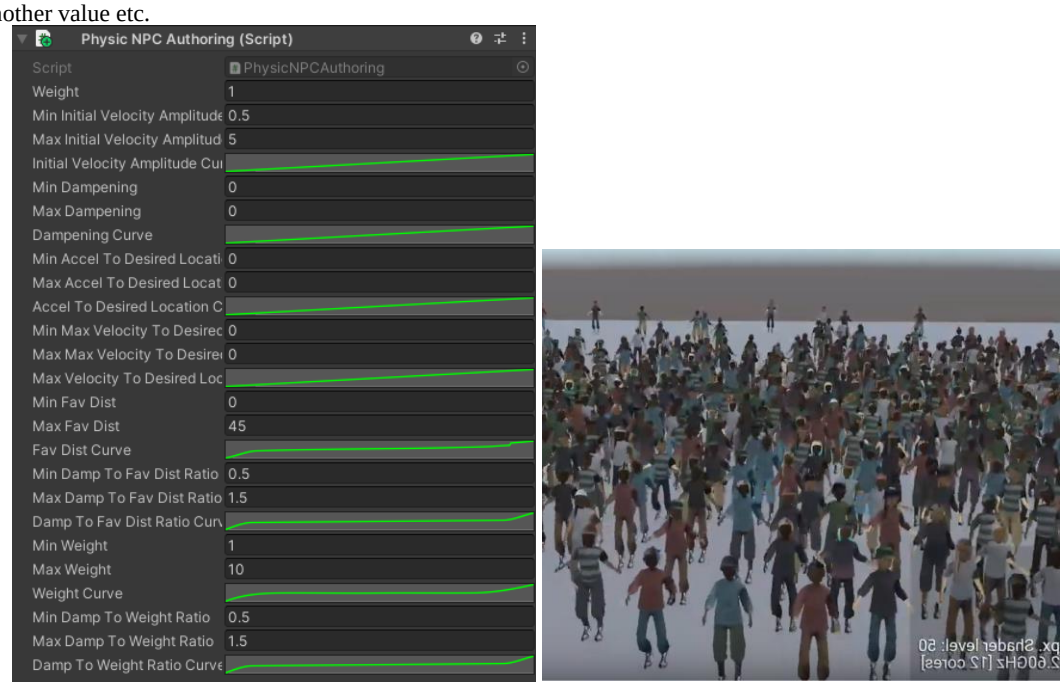

Figure 8: Settings of a realistic crowd and the result in game

This way we have heavy NPCs on the periphery of the crowd that dampens their movement a lot, and light characters that are pushed around a lot at the center of the crowd. This reproduces the behavior you can see in real life where at some point in the crowd people form a sort of "progressive barrier" between the chaotic mosh pit and the people watching the concert in a more calm and orderly manner.

4.4.1 Baking animation curves

Animation curves are not supported on Components, Managed or Unmanaged, so they are baked into arrays of float with a set number of values depending on how precise you want them to be. If a baked curve is too precise, the baking process will be very long and build and entering play mode will be very long.

Snippet 4: Baked

```
public struct BakedAnimationCurve
{
   public float[] Values;
   public int Precision;

   public static BakedAnimationCurve BakeAnimationCurve(AnimationCurve curve, int precision=1000)
```

```
  {
    var bakedCurve = new BakedAnimationCurve();
    bakedCurve.Values = new float[precision];
    for (int i = 0; i < precision; i++)
    {
      bakedCurve.Values[i] = curve.Evaluate((float)i/(float)precision);
    }
    bakedCurve.Precision = precision;
    return bakedCurve;
  }

  public float Evaluate(float t)
  {
    int index=Mathf.FloorToInt(t*Precision);
    if (index < Precision)
    {
      return Values[index];
    }
    return Values[Precision-1];
  }
}
```

5 HOW TO INTERACT WITH REGULAR GAME OBJECTS

There are two types of systems: the managed system, they are class derived from SystemBase, and the unmanaged system, they are struct that implements ISystem.

Only managed systems can interact with MonoBehaviours.

I added a MonoBehaviourObstacle component that can be added to any GameObject to make it interact with the simulation. On first launch it generates a GUID and registers itself to the system. If the GameObjects is static, it stops there, else it will update its position each frame it moves.
These obstacles are then known by the PhysicsNPCSystem and considered on each frame of the simulation.

Snippet 5: MonoBehaviourObstacle

```
public class MonoBehaviourObstacle : MonoBehaviour
  {
    [SerializeField] private float _radius = 1;
    private Guid _id = Guid.NewGuid();
    private Vector2 _cachedPosition;
```

```
    PhysicsNPCSystem _physicsNPCSystem;

    private void Start()
    {
      _physicsNPCSystem =
World.DefaultGameObjectInjectionWorld.GetExistingSystemManaged<PhysicsNPCSystem>();
      var twoDPosition=new Vector2(transform.position.x, transform.position.z);
      _physicsNPCSystem.RegisterObstacle(_id, twoDPosition, _radius);

      if (gameObject.isStatic) enabled = false;
      return;
    }

    private void Update()
    {
      Vector2 newPosition=new Vector2(transform.position.x,transform.position.z);
      if (_cachedPosition != newPosition)
      {
        _physicsNPCSystem.UpdateObstacle(_id, newPosition);
        _cachedPosition = newPosition;
      }
    }
  }
```

I then added code to the physics system so that these obstacles act as complex walls, each NPC that comes in contact with them will have its direction along the axis perpendicular to the "Obstacle to NPC" axis flipped.

6 CONCLUSION

DOTS is a fantastic set of tools for optimization and performance, and while still under development, its flexibility allows external developers to create very powerful addons.

The intrinsic chaos of a mosh pit was something difficult to simulate without hindering performance before DOTS existed, but now it is feasible to do.

REFERENCES

[1] Unity DOTS presentation page: https://unity.com/en/dots

[2] ECS documentation page on unmanaged components : https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/components-unmanaged.html

[3] .NET documentation page on unmanaged types : https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/unmanaged-types

[4] Burst documentation : https://docs.unity3d.com/Packages/com.unity.burst@1.8/manual/index.html

[5] Job System documentation: https://docs.unity3d.com/Manual/JobSystem.html

[6] Unity documentation page on Mecanim : https://docs.unity3d.com/462/Documentation/Manual/MecanimAnimationSystem.html

[7] Topic on Unity's forum created by the user "Iclemens" to discuss the different options for DOTS Animation: https://discussions.unity.com/t/dots-animation-options-wiki/894948

[8] Kinemation documentation:
https://github.com/Dreaming381/Latios-Framework-Documentation/tree/main/Kinemation%20Animation%20and%20Rendering

[9] ECS documentation page on Blob Assets : https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/blob-assets-concept.html

[10]

A APPENDICES

A.1 Demonstration videos

https://youtu.be/M71WioGfhFY : System with high dampening and high weight on the periphery, to contains the more chaotic individuals at the center.

https://youtu.be/2NvL48p4ucw : System with no dampening and no acceleration toward a velocity that would bring the NPC closer to its preferred position