FluidNinja LIVE Manual

Latest version:	1.7.4.1 for UE 4.26 - 4.27
	1.7.5.2 for UE 5

1.7 feature list: Chapter 29
1.7 playable demo exe: LINK
1.7 youtube playlist: LINK

NinjaLive is providing real time, responsive fluid simulation inside Unreal
For baking fluid simulation, see NinjaTools

Document updated: 7 July 2022

Support: andras.ketzer@gmail.com
kynolin@gmail.com

Discord:	https://discord.gg/rgEtwua2tu
Twitter:	https://twitter.com/FluidNinjaLIVE
Homepage:	unrealengine.com/marketplace/fluidninja-live

Document share URL:
https://drive.google.com/file/d/1I4dglPjeXLcNkSGxGok8sQCy59qgYcF9

Contents

1. Step by Step guides in this Manual

Topics	*Chapter*
Merging NinjaLive to your project	9
Adding NinjaLive Component To Your Own Actor Classes	10.3
Linking Niagara and Ninja	20.3
Cinematics Rendering via Movie Render Queue	23.3
Linking Volumetrics And Ninja	24.4
Reconfiguring a sim container to use 3D sim mesh	27.1
How to disable Dynamic Asset Lookup: in the FAQ doc	i005

Tutorials on YouTube

Topics	*URL*
Tutorial videos playlist	LINK
Showcase videos playlist	LINK
UseCase videos playlist	LINK
Introduction	LINK
Merging NinjaLive to your project	LINK
Changing default (planar) TraceMesh to Custom Mesh	LINK
Use TraceMesh to detect overlap (instead of InteractionVol.)	LINK
User defined line-tracing source (instead of Camera)	LINK
Render Buffers & Output Materials	LINK
Sim tiling / wrapping	LINK1, LINK2
Painting persistent Flow patterns (2D, Fog, Clouds...etc)	LINK
Using Sequencer to animate Objects and Sim Params	LINK
Character FX related videos:	
Adding NinjaLiveComponent to Your Character	LINK
SimDetails, CameraFacing, SingleTargetMode	LINK
Repositioning TraceMesh, adding Custom Sockets	LINK
Multiple fluidsim components in a single character	LINK
Hands: two dedicated sim components tracking all fingers	LINK

2. Introduction

FluidNinja LIVE is providing real time, responsive fluid simulation for PC and Console developers, inside Unreal. Following a 5 minute setup (see Chapter 9), responsive fluid simulation *Actors* could be placed on level. Live *ActorComponent* could be added to user defined Actors and drive other Components (eg. Niagara, VolumeSmoke, Fog). Live is easy to customize, features *Preset* based sim management and user editable *Output Materials*.

2.1 Inputs

Niagara Particles (position, velocity), StaticMeshes, PhysicsBodies (position, size and velocity), Skeletal Meshes (bone position and velocity), Textures and Materials (density, velocity) SceneCaptureCamera, Streaming video, User gestures (via mouse and touchscreen)

2.2 Level & game design

Live offers LOD, Proximity based Sleep/Wake and Memory Pooling - supporting a design approach with multiple sim containers placed on level and attached to characters.
Live 1.7 comes with complete World-Space support, driving Volumes, Niagara particles, Landscape Materials and Foliage - enabling users to drive all dynamic systems with fluidsim.

2.3 Rendering

Live uses 2D fluid simulation, combined with technology that makes it behave like a volume.

A. Camera Projection is using a *volume* to track objects in the sim area, mapping 3D positional and velocity data to the 2D sim using *Line Trace* from the camera - or from a user defined point.

B. ScreenSpace: in case we are running the sim on a camera facing plane, viewers could walk around the interaction volume, while the patterns generated by overlapping objects appear in the correct spatial position.

C. World Space: a 3D inertial frame of reference is supported. In case the sim area is moving (eg. attached to a pawn or vehicle), fluid density "lags behind", velocity and acceleration is properly handled. The moving direction of overlapping objects is also captured, tapping into simulation velocity.

D. HeighFields: sim density and pressure could be used to drive true 3D volumes as a height-map, while sim velocity drives the flow of volumetric noise. Alternatively, the height could be used to distort geometry (eg. water) or for Parallax Occlusion Mapping.

E. Raymarching: level placed light sources could generate dynamic self shadows on the 2D fluid or in the 3D volumes.

In the end, we have dynamic fluid in 3D - with the GPU demand of a 2D sim.

2.4 NINJALIVE vs NINJATOOLS

Both FluidNinja projects are focusing on fluid simulation inside UE. FluidNinja LIVE is for interactive fluidsim. FluidNinja VFX TOOLS is for *baked, non-interactive fluidsim*. The same visual aesthetics could be achieved with both projects - the main difference is, that TOOLS runs "standalone": it is an editor tool to generate game assets, it is NOT running together with the game and it is optimized to BAKE fluidsim to simple, looped sequences and materials that play these sequences.
LIVE and TOOLS could be used separately, or combined. For example: having a church interior with hundreds of candles definitely needs a baked solution - while having a flaming ball on the player character's hand or making a portal that interacts with the player might need a responsive sim solution.

3. Location & Size

NinjaLive is a compact Unreal Project - its feature set could be utilized in games by merging, *see Chapter 9*. It is based on standard Unreal assets (Blueprints, Materials), does not contain C++ code / pre-compiled elements / third party content. Cooking, compiling and packaging is tested for Windows PC, Apple PC, Android Mobile and iOS mobile. Compiled demo: LINK

Live blueprints are *annotated*, visually organized and cross referenced with links, to help developers understand and modify them. Example screenshots: BP1, BP2, BP3

The project is located in a single folder: /Content/FluidNinjaLive
Functional core is 30 Megabytes. Removable Tutorials 120 MB, UseCases 190 MB. The two most important files, *NinjaLive* (Actor class) and *NinjaLiveComponent* (ActorComponent class) are 2 MB and 7 MB, respectively.

4. Versions

NinjaLive is developed in separate branches to ensure compatibility with different UE versions.
Check the version of your NinjaLive project at: /Edit /Project Settings /Project /Description

UE 4.20 - 4.22 branch,	Latest NinjaLive version: 1.0.1.0
UE 4.23 - 4.25 branch,	Latest NinjaLive version: 1.6.23.0
UE 4.26 - 4.27 branch,	Latest NinjaLive version: 1.7.4.1
UE 5.02 branch,	Latest NinjaLive version: 1.7.5.2

5. Usage

NinjaLive is a standalone Project. As a first step, open it in Unreal Editor: check the Tutorial and Usecase Levels! When starting the game / pressing “Play”:
Use Play in “Selected Viewport” Mode -- do NOT use “Simulate” Mode
For running ninja in Standalone Mode, see: chapter 21.3.6 (LiveCompatibility)

In order to utilize fluidsim features in *your* project, NinjaLive should be merged:
Prepare the *target project* to host NinjaLive, by setting the directly specified settings:
See Chapter 9: Merging NinjaLive to your Project

Once merging is done, place/spawn NinjaLive fluidsim Actors at your levels - or add NinjaLive fluidsim Component to your own object classes. To make fluidsim Component work properly in your Actors, apply the directly specified Low Level Settings to the host actor class:
See Chapter 10.3

Optionally you could utilize additional tools like Preset Manager with custom GUI, Memory Manager, Utilities (See *Chapter 11*) and Interface Controller.

Fluidsim related parameters could be stored / loaded as *Presets* and handled by the Preset Manager (See *Chapters 6.4 and 21*).
Functional parameters (eg. LOD settings, interaction filters) could be set at the *Actor* and *ActorComponent* Details panel - as exposed variables - or set by blueprints using *Live Interface*.

Besides NinjaLive Managers, UE Sequencer could be used to control SimContainers.
See *Chapter 23.*

6. Core Functions

6.1 NinjaLiveComponent

FluidNinja Live is a library of functions built around a single, compact unit:
"*NinjaLiveComponent*", often addressed as "sim component", located in the project root.
It is an "ActorComponent" class object that should be added to a host Actor in order to use it.
The host is always called "owner". The sim component could run autonomously (loading its default preset, creating RenderTargets for itself) - OR - it could connect to the interactive *Preset Manager* and to the *Memory Manager*. An important restriction: sim component does _not_ have built in overlap detection volume. By default, Live Actor is feeding the embedded LiveComponent with information on overlapping agents. When LiveComponent is not receiving specific overlap information from the owner (eg. the owner is not a ninja Actor) , it is constantly tracking predefined owner components (eg. Bones, Sockets, StaticMesh Components...). Imagine the Component being added to a “Torch” Actor, constantly tracking the torch-head socket to generate fire at that point.

6.2 NinjaLive

"*NinjaLive*", an "Actor" class object, referred as "sim container" or "sim area", located at */Content/FluidNinjaLive* ... is a dedicated owner, embedding "*NinjaLiveComponent*" and adding two important features to its core feature set:

(A) activation volume (usually much larger than the sim area): the proximity of any user defined agent could switch the sim component between wake/sleep states.
(B) interaction volume (usually same size as the sim area): detecting overlapping / colliding objects and forwarding the spatial information + velocity to the sim component. Using the interaction volume, anything could interact with the simulation: *Static- and Dynamic Meshes, SkeletalMeshes, PhysicsBodies, Destructibles*.

6.3 Actor Details VS ActorComponent Details

When selecting an Actor on level, the *Details* panel is displaying all the exposed variables of that instance. In the case of "NinjaLive" Actor, the variables are collected into 4 groups, all starting with "Live" prefix: *Activation, Interaction, Debug, ComponentOverrides*.

IMPORTANT: *NinjaLiveComponent* settings are exposed on the component level - accessible only at the Component Details panel - NOT visible at the Actor Details panel!

Accessing Component Details:

Select *NinjaLive Actor* - or any other sim component owner Actor on level. At the Details panel, there is a top section, where Actor Components are listed (to see all components, this "top list window" should be rescaled). Select "*NinjaLiveComponent*".

Here is a visual guide on "how to select NinjaLiveComponent" ---> IMAGE LINK

Until the Component is selected, the Details panel displays its (the component's) properties - and NOT the owner properties. *NinjaLiveComponent* variables are collected to 9 groups, all starting with "Live" prefix:
Activation, MemoryManagement, Performance, Compatibility, Debug, Generic, Interaction, BrushSettings, Raymarching

In this document, we are referring to the two states of the Details panel as "Actor Details" and "Component Details".

6.4 Preset Manager and Presets

Ninja simulation parameters (eg. speed, forces, inputs) are stored in PRESET files.
PRESET MANAGER is an *editor only tool* with a custom UI, made to...

A. develop / tweak fluidsim parameters in real-time
B. load / save settings as preset files

PresMan. is a *helper* function, *not* needed to run ninja at all. In case you'd like to control sim via *game logic* like Preset Manager does, read *Chapter 26.: "Controlling Live in real time"*.

6.4.1 A level placed NinjaLive container is associated with a *Default preset* at:
NinjaLive Actor Details /NinjaLiveComponent /LiveGeneric /DefaultPreset
Fluidsim params stored in the *Default Preset* are loaded when a sim container initializes
(No PresMan needed, containers are autonomous!)

6.4.2 Preset Manager could be used by placing */FluidNinjaLive /NinjaLive_PresetManager* blueprint actor on a level. The most important UI elements are (see more at 21.1):

A. In the top row, you could enable *Tooltips* (recommended!)
B. In the second row (printed in red) you could SELECT a given NinjaLive Actor
C. In the third row (printed in blue) you could LOAD a preset for the selected actor.
   The default preset of a given NinjaLive Actor is marked with * symbol
D. Using the floppy disk icons, you could save and duplicate presets

6.4.3 More *Preset Manager related topics* in this Manual:

10.2.2	*Preset Usage*
14.1	*Remove Preset Manager From Levels Before Compile*
21.1	*GUI exposed params*
21.3.2	*LiveGeneric /DefaultPreset*

6.4.4 Preset Manager Error Handling

A. Level-placed PresMan could be de-activated, at PresMan Actor Details Panel.
Icon turns gray (when enabled, it is black).

B. PresMan tries to reach clients at startup. You could define the default client at the PresMan Actor Details panel. In case the default client field is empty / or client is not present / or client is asleep: PresMan tries to load someone else. Once started, you could manually select which client (which NinjaLive Actor) you would like to load for tweaking.

C. Multiple Preset Managers per level: DO NOT

D. PresMan can't find clients - case1:
Preset Manager could go blank when it can not connect a valid ninja container. A possible cause: containers are usually set to be proximity activated. When the pawn or spectator camera (that triggers the proximity sensor) is away, the container is inactive (asleep). PresMan can _NOT_ wake sleeping containers. This is communicated via an explicit on screen message.

E. PresMan can't find clients - case2:
PresMan recognizes clients via the Live INTERFACE -- in case you are not using the standard NinjaLive BP - eg. you have embedded NinjaLiveComponent to your custom BP, like a pawn - make sure you have added "Live Interface" under BP/Class Defaults/Interfaces

F. PresMan can't find presets:
NinjaLive actors contain a non-dynamic reference for preset LOOKUP location.
By default, it is: /Content/FluidNinjaLive/Presets
In case you have moved ninja to a new subfolder: update the "*PresetSearchPaths*" variable in *NinjaLiveComponent* Blueprint.

G. Multiple NinjaLiveComponents per Actor:
Preset Manager is currently *not* prepared to handle multiple NinjaLiveComponents embedded in a Single Actor. You could try developing presets for each component separately (using duplicated actors) - and when done, simply referencing the final preset files for each Component under LiveGeneric/DefaultPreset.

6.5 Simple Painter Mode

Briefly: track objects and draw trajectories without running fluidsim (eg. footsteps)!
Mode switch is located at: *NinjaLiveComponent /Live Interaction /SimplePainterMode*
Using the *EnablePaintBufferOffsetInWorldSpace* switch, we allow ninja to use SimplePainter IN WORLDSPACE. For practical examples: see Tutorial Levels 7, 32

NinjaLive has the potential to track objects, detect collisions and write the data to a RenderTarget - this is what we call "Collision Painter". The function could also accumulate and erode paint strokes by time.
The built in switch - SimplePainterMode - (a) discouples Collision Painter from the rest of the system and (b) disables NinjaLive fluidsim functions. The result is a minimal system using only 1 or 2 RenderTarget (depending on WorldSpace usage), keeping GPU load on a very low level. Combined with the user editable Output Materials feature, SimplePainter could be used efficiently for certain VFX types (eg. feed particle fields with velocity, print footsteps)

Blueprint link: see MODULE061 A-H in *NinjaLiveComponent* Blueprint

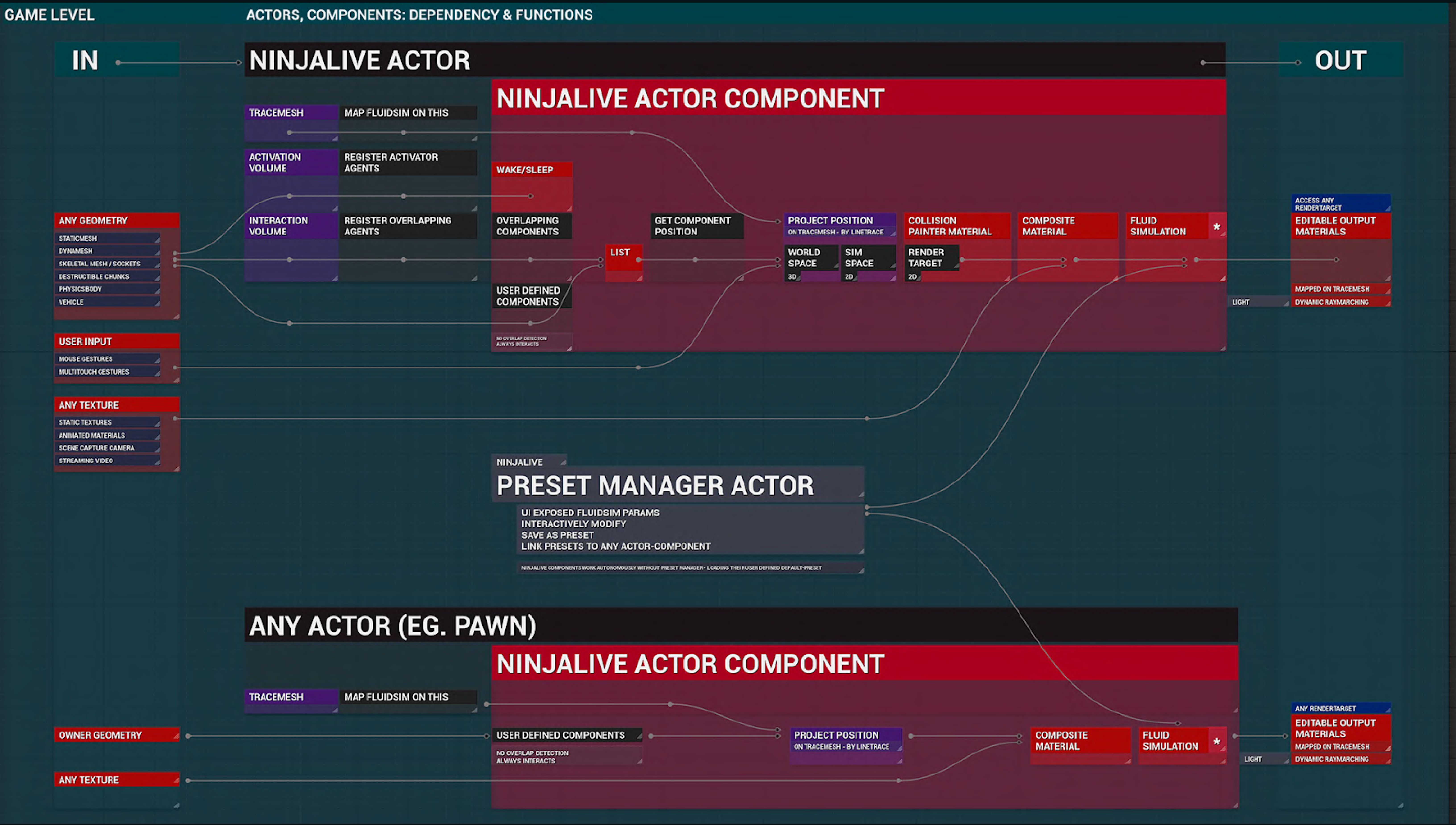


Chart 1 Actors, Components: dependency and functions. High.res bitmap: LINK

7. Memory Demand

NinjaLive allocates memory mainly for RenderTargets. The rendering pipeline uses 8 RenderTargets for a single simulation container by default: 1 four-channel [RGBA, for CollisionPainter], 5 single channel [R, for Density and ScalarFields] and 2 bi-channel [RG, for VectorFields] - all set to 16 bit precision. The pipeline could be reconfigured many ways (resolution, bit depth, channel usage).

A 256x256 fluidsim container with default settings allocates 1664 Kilobytes of memory - this could be crunched or expanded, depending on the needs. Similarly: 512x512 area = 6656 Kbytes, 1024x1024 area = 26624 Kbytes, 2048x2048 area = 106496 Kbytes. Of course, the sim area could be non-square (eg. 128 x 512), mem consumption is changing accordingly.

Link1: See Help.uasset /Chart2 /RenderTargets for more details on possible RT configs.
Link2: Select any actor containing the sim component, and switch on "/Component Details /LiveDebug /ShowMemoryManagement" for dynamic, on-screen report on actual memory consumption of a given container.

8. Performance (excerption)

Important: Live 1.3 has been released with major performance improvements. The new results are NOT YET included in Chapter 12 / the currently available data is outdated.

The following exceptions (typical and peak) are here to represent the magnitude of changes.

TYPICAL PERFORMANCE

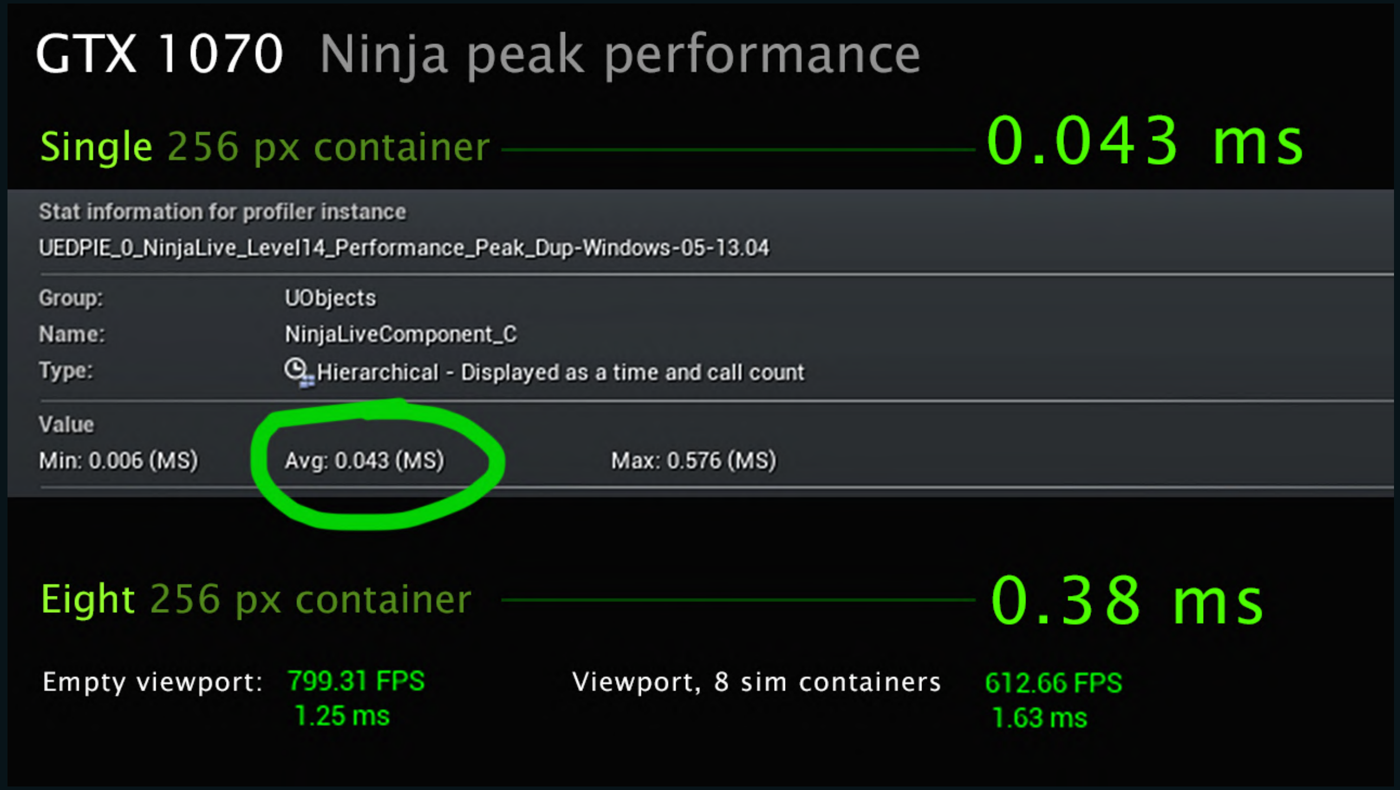
LIVE 1.2 vs 1.3
GTX 1070, UE 4.23, average FPS

Level 11: SmokeChamber test scene FOUR 720 px fluidsim containers
Live 1.2: 140 FPS vs Live 1.3: 230 FPS

Level 13: SmokeChamber test scene TWENTY 720 px fluidsim containers
Live 1.2: 100 FPS vs Live 1.3: 190 FPS

PEAK PERFORMANCE

LIVE 1.3 (unlit scene, antialias and postprocessing off)
GTX 1070, UE 4.23


GTX 1070 Ninja peak performance

Single 256 px container 0.043 ms

```
Stat information for profiler instance
UEDPIE_0_NinjaLive_Level14_Performance_Peak_Dup-Windows-05-13.04
Group:              UObjects
Name:               NinjaLiveComponent_C
Type:               Hierarchical - Displayed as a time and call count
Value
Min: 0.006 (MS)     Avg: 0.043 (MS)          Max: 0.576 (MS)
```

Eight 256 px container 0.38 ms

Empty viewport: 799.31 FPS 1.25 ms
Viewport, 8 sim containers 612.66 FPS 1.63 ms

9. Merging NinjaLive to your project

Before merging, the target project needs to be prepared!
NinjaLive requires the following Project Settings to work properly:

STEP 1: *add custom Trace Channel*

NinjaLiveComponent uses a custom channel - *FluidTrace* - to perform line tracing while projecting WorldSpace collision coordinates on the fluid simulation mesh (*TraceMesh*).

A. Go to */Edit /Project Settings /Collisions /Trace Channels*
B. Choose "*Set New trace channel*"
C. Name = *FluidTrace*, Default Response = *Ignore*

Note 1 : *Trace Channel Index* does *not* matter - "FluidTrace" could be added to an existing set of Trace Channels, no need to re-configure the order of channels.
Note 2: As a consequence of the default "ignore" response, all objects will be transparent for NinjaLive line tracers - except dedicated *TraceMeshes* in *NinjaLiveComponent* owners.

STEP 2: *enable "UV from Hit" engine feature*

Go to: */Edit /Project Settings /Engine /Physics /Optimization* and enable the "*Support UV From Hit Results*" option. Collision data is projected on the *TraceMesh* by linetrace as "Hit". Using this feat, 3D Hit coords could be converted to 2D sim UV-space instantly.

Please double check the above steps - compare your editor to this screenshot: LINK

STEP 3: *merging*

A. Quit Unreal. Copy "*/Content/FluidNinjaLive*" subfolder from the original (non-merged) Live project to the *target* project root (*/Content*) using Windows or iOS file manager, while UE is *not* running.
Emphasis: *do not* copy the *whole* NinjaLive project structure to the target project - copy *only* the "FluidNinjaLive" subfolder located in the "Contents". *Do not* try to use Unreal *Migrate* or *Import* functions - *copy* the above described subfolder.

B. Open *target* project in Unreal Editor. Open *NinjaLive Blueprint* with blueprint editor.
Location: */Content/FluidNinjaLive/NinjaLive.uasset*
Press "*Compile*", then "*Save*".

C. Open *NinjaLiveComponent Blueprint* with blueprint editor.
Location: */Content/FluidNinjaLive/NinjaLiveComponent.uasset*

Set the default value of "*TraceChannel*" and "*CollisionChannel*" variables to "*FluidTrace*"
See this visual guide on blueprint editing: LINK

Press "*Compile*", then "*Save*".
In case the blueprint compiles without errors, we are almost done.

D. In the level editor, select any *NinjaLive Actor* on a level. Select *NinjaLiveComponent* (see HOW). Check the "*Live Compatibility*" group: the top 3 input fields should be set to "*FluidTrace*".

Once done with these steps and checking: PROJECT MERGING IS FINISHED

10. Settings

10.1 Ninjalive Actor Parameters (briefly)

Chapter 21.2 is providing a full description on all *NinjaLive Actor* parameters.
Right here, we are covering only a few *key* settings.

Once the original NinjaLive project is opened - OR - merging NinjaLive to another project is finished, *NinjaLive Actor* (sim container) is ready to be deployed to game levels.

(1) drag-n-drop NinjaLive from the Content Browser to a level - OR - (2) copy-paste already configured Actors from NinjaLive tutorial levels to your project levels - and modify them to get satisfactory results. Start experimenting with the settings, try to understand how Actors on tutorial levels are set up. You can find tutorial videos in this playlist.

When selecting a "NinjaLive" Actor on level, the Details panel is displaying exposed variables, collected in FOUR groups

LiveActivation: [9 params], disable switch + wake/sleep related params (activator, range...etc)
LiveInteraction: [12 params], scale sim area + filter interaction types by class or bone name
LiveDebug: enable on screen status messages
LiveComponentOverrides: helper function to set variables when multiple actors are selected.
See NinjaLive blueprint / GROUP1 / SUBGROUP003 for more information.

10.1.1 DISABLE/ENABLE

NinjaLive actors could be "force deactivated" using the "Disable" flag
(a) Select one/more "NinjaLive" Actor on level
(b) find "LiveActivation" param group on the Details panel
(c) set Disable flag state: on/off

The disable switch OVERRIDES activation settings: the Actor remains passive even if the activator is in range. Once disabled, the on-level Actor ICON turns gray (originally red).

10.1.2 SCALING / SIZE

Do _not_ try to scale the sim container actor via viewport transform gizmo - nor using the numeric *Scale Transform* input at the Actor's Details panel. Find the "Live Interaction" param group at the Actor's Details panel, and set "Trace Mesh Size" and "Interaction Volume Size" to influence the size of fluid simulation plane and the size of interaction volume, respectively.

The default *TraceMesh*, found in *NinjaLive Actor,* is a planar mesh. For this reason, when you set "Trace Mesh Size" on the details panel, the Z-component of the input field does not have any effect on the visuals and workings. Later on, you will learn how to add a custom *TraceMesh* to your own actors. In case these meshes are not planar (eg. a hemisphere), the Z-scaling component starts to make sense - and that is the reason for that param being there.

The Interaction Volume is a true 3D object by default, so all dimensions of the "Interaction Volume Size" param field have effect on the workings.

Note: *TraceMesh* and InteractionVolume both have a separate visibility flag (at "Live Interaction" group). Make sure that these flags are on - to visualise the mentioned geometries. The two params are: "ShowTraceMeshInEditor" and "ShowInteractionVolumeInEditor"

10.1.3 NON-UNIFORM SCALING

Using the above mentioned options, you could set up a rectangular (non-square) *TraceMesh*/ simulation plane. Since fluidsim XY resolution is also manually provided (non-automatic), you should make sure that side-proportions match - eg. a *TraceMesh*, sized X=4,Y=8 is matched with a Fluidsim resolution set to 256x512 or 512x1024.

Note: you could reach fluidsim resolution settings at the Details panel of *NinjaLiveComponent*. Select *NinjaLive Actor*. At the Details panel, there is a top section, where Actor Components are listed (sometimes, to see all components, this "top list window" should be rescaled - by dragging the bottom part of the window-handle downwards). Select "*NinjaLiveComponent*". Until the component is selected, the Details panel displays its (the component's) properties - and NOT the owner properties.
Find "Live Performance" submenu, set "Resolution X/Y" respectively.

10.1.4 POSITION & ROTATION

As opposed to Scaling: Position and Rotation of the sim area should be set using the viewport gizmo or the Transform input field at the Actor's Details panel.

10.1.5 LOCALISING INTERACTIONS: OVERLAP PROJECTION

Params related to this section are located at: Actor Details /Live Interaction
While our simulation is 2D, we would like to seamlessly integrate it to 3D space - so, we came up with the "Overlap Projection" technique. Our fluid simulation container actor, placed on level has two related components:

(A) *InteractionVolume*: used to detect overlapping / collision - objects inside this volume could interact with the simulation - initial and ongoing overlap events are continuously monitored (both entering and leaving the volume). The type of interactions could be defined / filtered by the user (see next chapter)

(B) *TraceMesh*: we are mapping the simulation output to this mesh (by default a plane, but could be defined by the user) plus: we are going to perform a line trace against this mesh.

The sim container is triggered by objects that overlap with the Interaction Volume. Once overlapping object(s) / components are targeted, a line-trace is performed in their direction (A) from the camera or (B) from a user-defined point - against the *TraceMesh*. Line-trace hits the *TraceMesh*, the hit information is transformed to *TraceMesh* UV space (which is also simulation UV space), and the simulation density/velocity inputs are feeded with the related positional/velocity/size data.

Velocity could be derived from Frame/PreviousFrame positional delta or alternatively by calling (requesting from the engine) the world space velocity of the colliding object and transforming it to sim space. NinjaLive uses both methods. What if the motion is perpendicular to the *TraceMesh*? NinjaLive contains a certain processor to handle this, by increasing pressure and velocity noise at the given location, giving an illusion of an object "blasting through" a surface, producing shockwaves (if velo is large).
Non-perpendicular velocity vectors cause "drag-like" advection. We always check composite velocity, and add "blast factor" proportionally to the perpendicular component.

Now, imagine that we are forcing *TraceMesh* to be camera facing. So we are mapping collisions from the camera view to a camera facing plane. And remember: we are tracking objects / bones that are inside InteractionVolume, but not aligned on *TraceMesh* surface (imagine the limbs of a character or the debris of a crumbling wall): since trace-targets are moving in 3D space and the advection / density they generate is always positionally following them, the fluid seems spatially extended.
More on Camera Facing: 10.2.5

10.1.6 FILTERING INTERACTIONS

Params related to this section are located at: Actor Details /Live Interaction - allowing you to specify / filter interaction types for the simulation area.

/Overlap Filter Inclusive Object Type: an array, where you could add/remove object type-classes: WorldStatic, WorldDynamic, Pawn, Vehicle, Destructible, PhysicsBody
/Auto Exclude Large Overlapping Objects: eg. an extended floor object that overlaps the Interaction Volume, and we do not want it to interact with the fluid. You could do the same manually , under:
/Exclude Specific Actors From Overlap

Bone names: in case you have provided a skeletal mesh class (eg. Pawn) in the Filter array, the container is trying to detect all bones by default - most of the cases this is not what we want. "Partial name filter" allows you to type in "foot" ot "hand" - and all bones containing these strings will be included. Performance-wise, the most optimal solution is providing "Exact Bone Names": eg. "feet_r", "feet_l" for a Mannequin, walking in a puddle sim.

10.1.7 WORLD SPACE VELOCITY

There is a single param for channelling World Space velocity to the simulation: eg. we have a kettle full of swirling liquid, and we start to push the kettle / or we have a pawn with flaming hands, and he starts to run, and we expect the smoke/flames to be left behind.

The param that influences this phenomena is a "preset" param, that could be changed using the preset manager,
see "Velocity Field influenced by sim area motion" on the GUI - or modified by manually editing the preset file (defined at Component Details / LiveGeneric) at "VeloFromSimAreaMotion" field.

10.2 Ninjalive *Actor Component* Parameters (briefly)

Chapter 21.3 is providing a full description on all *NinjaLive Component* parameters.
Right here, we are covering only a few *key* settings.

Select any *NinjaLive-Component-Owner* Actor on level (eg. "NinjaLive" Actor or Orb, Pawn).
Select *NinjaLiveComponent** at the Actor Details panel (see *Actor Components* list at the top)

*Here is a visual guide on "how to select NinjaLiveComponent" ---> IMAGE LINK

Notice: as you select the actor component, the "Details" panel changes and you could see the exposed *Component Variables*, collected in 9 groups, all starting with "Live" prefix.
Listed in order of importance:

Interaction [11 params] - continuous (non-overlap based) interaction AND single target mode
Generic [14 params] - PRESET and INPUT / OUTPUT MATERIAL definitions + more
Performance [19 params] - resolution, LOD, render pipeline settings
MemoryManagement [3 params] - in case mem.manager is placed on level: set up connection
Raymarching [16 params] - picking lightsource for raymarching, setting params
Compatibility [6 params] - system level switches, eg.: Flip RenderTargets for Mobile compile
BrushSettings [11 params] - collision painter brush size overrides and noise settings
Activation [1 param] - component level disable switch (actor level is preferred, when available)
Debug [10 params] - enable on-screen reporting of LOD, memory...etc

10.2.1 DISABLE/ENABLE

When talking about *NinjaLive Actor*, we have TWO disable switches available:

Actor Details /LiveActivation /Disable	(preferred)
Actor Component Details /LiveActivation /Disable	(not preferred)

Normally, we disable systems on the Actor Detail level. Once *NinjaLiveComponent* is added to a random owner (eg. pawn), the high level (actor level) disable is not there anymore. In this case, the component level disable could be used.

10.2.2 PRESET USAGE

Component Details /Live Generic /Default Preset - this is where you could associate a default preset for a *NinjaLiveComponent* - this preset will be automatically loaded at area initialization. Note: in the Preset Manager, you could load / try arbitrary presets for a given area - but you have to provide DEFAULT here, at the Component Details.

10.2.3 OUTPUT MATERIAL USAGE

Output Materials: NinjaLive rendering pipeline is a chain of Blueprint controlled Materials, writing to RenderTargets. Any RenderTarget could be used as output, depending on the needs (typically Density, Velocity, Pressure and PainterInput). So, a typical "Ninja Output" material contains at least one "Parametric Texture Object" with Specified name, eg. "DensityBuffer", "PressureBuffer"...etc. these materials are tapped into the rendering pipeline, and use the needed raw sim output to create arbitrary shaders.

Note: by default, the render pipeline is processing MONOCHROMATIC density data, and the output is COLORIZED - based on the available data (eg. Tonemapping monochrome density with two colors). The density could be sent to a raymarch shader or Pressure used to perform WorldPositionOffset, Divergence for Refraction... really up to you. Have a look at the provided output material examples at: */Content/FluidNinjaLive/OutputMaterials/*

Below the array of output materials, there is an INTEGER input field, labeled as:
“Output Material Selected” - provide here the array index for your default/favourite material.

The array is also accessible in the Preset Manager - so you could check variations in runtime.

Btw, runtime: you could start/leave open the standard Unreal Material Instance Editor UI and tweak output material params while Ninja is running - and you see the real time fluidsim in the main viewport - very efficient way to fine-tune params!

10.2.4 RAYMARCHING

To set up *NinjaLiveComponent* for Raymarching, check *these* settings at *Component Details*:

- LiveRayMarching / Enable Raymarching switch
- LiveGeneric / OutputMaterials array: make sure there is a raymarch capable output material on the list
- LiveGeneric / OutputMaterialSelected: set the array index for the above material

10.2.5 INTERACTIONS

Component Details /Live Interaction: these settings are partly described in the next chapter, see: STEP 9-10, Setting up "non-overlap based" or "continuous" interaction with the owner.

Note: Continuous Interaction should be flagged ON every time you include Live Component to a new owner (the new host is NOT *NinjaLive Actor*, but some other class) - this flag ensures that LiveComponent could interact with the new owner. Other settings in this submenu:

Single Target Mode: focusing on a single superfast target (eg. pawn fist)
Character animation could be erratic, extremely fast movements occur, the temporal sampling of the collision painter simply can not cope with this, we see density dots. This mode focuses on a single target, and connects the sampling points via interpolation, creating a continuous motion-trajectory, drawing density and velocity lines between the sparse sampled position data points - this way making "fluid ribbons" / "smoke or flame trails" behind the superfast moving object. The target is the first item in the "Bone names'' list - or the first element found based on type filtering.
Using the "Single Target Set Sim Speed" flag, a bullet-time like "time dilation effect" could be achieved: the moving speed of the single target influences fluidsim play speed.

Camera Facing: performed by a generic function, stored at:
/Content/FluidNinjaLive/Core/NinjaLiveFunctions - usually called by the sim component
(see MODULE036 in the blueprint)

Important Note: when using NinjaLive as a component IN MOVING AGENTS (eg.: a Pawn), keep Camera Facing in the sim component switched OFF and call the "Camera Facing" function in the owner Blueprint's Tick flow. This results in smoother rotation. Reason: the ticking of Pawn transform and *TraceMesh* transform should be synchronized (in case of non-moving owner sync does not matter).

See included "motile owner" type blueprints - referred at 10.3

10.2.6 USE CUSTOM LINE TRACE SOURCE (instead of camera)

New feature, added to NinjaLive 1.2 ---- UI access:

NinjaLiveComponent Details Panel /LiveInteraction /UseCustomTraceSource
NinjaLiveComponent Details Panel /LiveDebug /VisualizeCustomTraceSource

Levels demonstrating the feature: LEVEL 2C -- LEVEL 21 Stage 6B -- LEVEL 24A, 25, 26
Tutorial video on feature: LINK

NinjaLive uses line tracing to project 3D collision data to the 2D simulation plane. By default, the trace line starts from the camera and targets the overlapping / colliding object. At the point where the trace line intersects the TraceMesh, the collision painter is drawing to the collision buffer. This is how overlapping objects generate density/velocity for the simulation.

Using the new feature, we could define a custom line tracing source point,
by adding XYZ offset to the sim container position.

Continuing 10.2.6

Advantage 1: no visual artefacts when looking at planar surfaces from low angle ("FPS view")

Advantage 2: detaching simulation container and simulation output - running the sim in the background while the user and the camera is somewhere else

> Note1: before adding this new feature, NinjaLive was already capable to run "non-interactive" simulations in the absence of camera, eg. using a texture or material to generate density while ignoring overlapping objects --- see Level 24B setup!
>
> Note2: to perform the "run a sim while we are away" feature, we need to switch off LOD and "Pause sim when not visible" --- see Levels 24A,24B,25,26 for details!

Advantage 3: no impact doubling when running the simulation on hollow meshes and viewing "from the other side". Imagine a sphere, used as TraceMesh - and an object colliding to the RIGHT hemisphere. In case we watch the impact "from the other side" the trace line (starting from the camera) crosses the LEFT hemisphere as well, generating UNWANTED simulation input. In case we are NOT using the camera as trace source, this could be avoided --- see Level21, Stage 6B!

Advantage 4: we could avoid VR artefacts - when the user is rapidly changing his point of view and the overlapping object "hovers" above the TraceMesh (not exactly intersecting the tracemesh), the projected collision point wanders (like a shadow of an "above the floor object" while you are moving the torch). By using a fixed line trace source point, this could be avoided.

10.3 Adding NinjaLive Component To Your Own Actor Classes

We are going to update an existing actor class to contain *NinjaLiveComponent*. Typical cases include Pawns, Vehicles and other motile agents.

Before start, we'd like to mention: on the NinjaLive tutorial levels you'll find multiple examples for this setup - including a floating orb and the classic UE-Mannequin pawn, both with *NinjaLiveComponent* added. Blueprints could be found at these locations (folders):

/FluidNinjaLive/Tutorial/BP_NinjaDemo_MovingBall1_NinjaComponentAdded
/FluidNinjaLive/Tutorial/UE_Mannequin_UsageExamples/ThirdPersonCharacter_NinjaAsComponent

Important: *NinjaLiveComponent* does _not_ have built in overlap detection functionality, interacts only with predefined (user defined) owner components (eg. Bones, Sockets, Meshes). This type of interaction is labeled "non-overlap based" or "continuous" interaction, handled by MODULE021 in the *NinjaLiveComponent* blueprint. Below, you'll find instructions on how to manage this (STEP 8-9, explained)
Let us start with a quick step-by-step guide, followed by explanation on the non-trivial steps:

1. open owner (the planned sim component host blueprint)
2. add component: *NinjaLiveComponent*
3. add component: *TraceMesh* (StaticMeshComponent)
4. set the value of "TraceMeshComponent" variable of *NinjaLiveComponent* (value = *TraceMesh*)
5. set *TraceMesh* properties (note: scale X,Y is related to sim X,Y resolution!)
6. add NinjaLiveInterface to the owner blueprint (at Class Settings /Interface)
7. set *NinjaLiveComponent* default properties: while remaining in the blueprint editor, select *NinjaLiveComponent* in the "Components" list of owner to access *NinjaLiveComponent* Details panel. Find "Live Interaction" parameter group
8. set the "Continuous Interaction with Owner Actor" switch to "ON"
9. set continuous Interaction Inclusive Object Type / Bone names
10. set the default value of all other params here (in the blueprint) - eg. default preset, material
11. enable *TraceMesh* CameraFacing (optional)
12. save blueprint, and drag it on level from the Content Browser

STEP 3, ADD TRACEMESH COMPONENT TO OWNER

All *NinjaLiveComponent* owners must contain a StaticMeshComponent labeled as "TraceMesh". *TraceMesh* has two functions: linetrace is performed against it - and the fluidsim is mapped on it. Depending on the context, we could call this mesh "simulation plane" or "tracemesh" - it is the same.
Any mesh would do - NinjaLive contains a simple StaticMesh type plane as default object, add that one for start: /FluidNinjaLive/NinjaLiveTraceMesh

STEP 4, TELL NINJALIVE ABOUT THE TRACEMESH

NinjaLiveComponent contains a "*TraceMeshComponent*" variable that MUST be set by the owner. Please have a look at this blueprint:
/Content/FluidNinjaLive/Tutorial/BP_NinjaDemo_MovingBall1_NinjaComponentAdded

OnEventBeginplay, set *TraceMesh* as "value" to "*TraceMeshComponent*" variable of *NinjaLiveComponent*. Steps:
a. pull *NinjaLiveComponent* to the blueprint editing-surface
b. pull a data-flow thread from the component, and type: "*TraceMeshComponent*", place a "set" node
c. pull *TraceMesh* to the blueprint editing-surface - and wire it to the above "SET" node.
This way, we are explicitly telling *NinjaLiveComponent* about the *TraceMesh*, contained by the owner. From now on, the component does a lot of things with the mesh, all automatically.

STEP 5, SET TRACEMESH PROPERTIES

a. Editor Visibility could be switched off: *TraceMesh* component details /Rendering / Visibility
b. Collision settings are extremely important - but *NinjaLiveComponent* handles this automatically by force setting all the necessary params: see MODULE004 in the sim component.
One thing that could _not_ be set from blueprint: ***double sided***. This property should be enabled at the Mesh Editor. For our ready-to-use plane it is flagged "on''.
c. Scale: three methods to set *TraceMesh* scale
- At the level placed instance - by selecting the component (this effects only that instance)
- In the owner blueprint, selecting *TraceMesh* and set default transform (effects all instances)
- Via an exposed variable (*NinjaLive Actor* does it this way, so you could customize instances)
Important: scale is related to simulation X,Y resolution - see: 10.1.3 NON-UNIFORM SCALING

STEP 6, ADD INTERFACE

Add *NinjaLiveInterface* to the owner blueprint at: *Class Settings / Interface*

NinjaLiveComponent-Containing-Actors (owners) should have this interface implemented...

a. to be visible/accessible for GUI-controlled preset management
b. to be excluded from the line tracing of OTHER owners
c. to access / modify exposed variables of any level placed sim component owner

Both *NinjaLiveComponent* and *NinjaLive_PresetManager* identify / collect "clients" based on INTERFACE (Get all Actors with Interface).

See: *NinjaLiveComponent*: MODULE020 (exclusion from linetrace) / NinjaLive_PresetManager: SUBGROUP003
On Tutorial Level 5 you could find a setup where NinjaLive_InterfaceController is communicating with a *NinjaLive Actor* using the interface, initiating a "fade off / shutdown" sequence, have a look! The clients are defined in the Actor Details/ Interface Target Actors array of controller.

STEP 8-9, SETTING UP "NON-OVERLAP BASED" OR "CONTINUOUS" INTERACTION WITH THE OWNER

Next, we are going to set *NinjaLiveComponent* default properties. While remaining in the blueprint editor, select *NinjaLiveComponent* in the "Components" list of owners - to access *NinjaLiveComponent* Details panel.
At the Component Details panel, find the "Live Interaction" parameter group. Set the "Continuous Interaction with Owner Actor" flag to "ON".
When a sim component is embedded to its native host - *NinjaLive Actor* - the owner is doing overlap detection and telling the component's line tracer "who to track". Sim component does not have built-in overlap detection functionality. When embedded to new owner (eg. pawn), line tracer falls back to a user defined list on "who to track" - the list contains type filters (StaticMesh, Dynamesh, Pawn...etc) and bone/socket lists ("Inclusive Object Type" and "Bone names"). This is all very similar to the Interaction settings of the Actor Detail menu - see 10.1.6 FILTERING INTERACTIONS for a general description on interaction filtering and bone names.

STEP 10:

Set the default value of all other params - eg. default preset, material
We have been talking about this in the previous chapter. See:
10.2.2 PRESET USAGE, 10.2.3 OUTPUT MATERIAL USAGE
While the brush size can be adjusted via Preset, you may want to tweak it in: Component Details /LiveBrushSettings

STEP 11: (OPTIONALLY) ENABLE TRACEMESH CAMERA FACING
at: Component Details /LiveInteraction

Please note: with rotating/motile agents, it is better to call the Camera Facing function in the owner blueprint, see 10.2.5 INTERACTIONS / Camera Facing for case description.

10.4 Other Ways To Include/embed/add Ninjalive To Actors

Adding *NinjaLiveComponent* to an object class is the preferred / advised way to equip any agent/object with fluidsim. However, there are other methods to do this. The updated version of this document will feature a chapter describing these methods. Briefly:

A. CHILD ACTOR COMPONENT

Unreal Engine implemented a structure not too long ago, called "ChildActorComponent". The structure enables us to add "Actor" class objects as components to another class (as opposed to adding "ActorComponent" class components). The UE implementation looks somewhat unfinished and accessing child parameters is a bit confusing - but we can add *NinjaLive Actor* with all its overlap and proxy activation management systems to any actor. We have managed to successfully embed *NinjaLive Actor* to the default Mannequin Pawn, see: *ThirdPersonCharacter_NinjaAsChildActorComponent* in this folder: */Content/FluidNinjaLive /Tutorial /UE_Mannequin_UsageExamples*

B. SPAWNING

NinjaLive Actor could be spawned by code / blueprints - and attached to the spawner, creating a similar situation as if it was an "owner". The main problem is, how to (pre)define the params of the "to-be-spawned" Actor? Two ways to do this:
(a) spawn, set params instantly, re-initialize the spawned Live actor
(b) by flagging "Expose on spawn" ON for variables in the "to be spawned" Blueprint (in this case NinjaLive), we could access them / set them before spawning - on the spawning node.
We have tested this as well - see: *ThirdPersonCharacter_NinjaSpawnedAttached* in this folder: */Content/FluidNinjaLive /Tutorial /UE_Mannequin_UsageExamples*

10.5 Interaction: setting up Objects to trigger NinjaLive response

---> See Tutorial Level 21 / Stage2 for examples
NinjaLive is pre-configured to handle a wide range of interactions as "recipient" - but some steps should be taken on the "causer" side as well (causer = collider, overlapping object).

A. OVERLAP

In case you want an agent to trigger *NinjaLive Actor*'s OVERLAP sensor:
tick ON the "Generate Overlap Events" option at Details /Collision (by default, it is OFF)
Repeating: by default, all build-in Unreal objects (eg. a sphere, a plane) are set "not-to" generate "overlap events", this setting should be changed.

> An interesting fact [UE bug? :)] - in case an object is INITIALLY (at BeginPlay) in the overlap zone (inside the interaction volume) of *NinjaLive Actor*, it will be detected as overlapping agent - the “generate overlap” setting influences only "latecomers" and everybody who wants to "leave". In case you have an agent that (1) starts in the area and (2) later leaves and (3) "GenerateOverlapEvents" is NOT enabled - NinjaLive will not receive a notice on the leaving and this might lead to anomalies.

B. SKELETAL MESHES

NinjaLive starts to track SkeletalMeshes if the "Pawn" interaction filter is active/added. What if a SkeletalMesh is not a pawn? Eg.: you place a SkeletalMesh class object to level and set it to play an animation, and it is not embedded in a blueprint? Not a problem: set the SkeletalMesh (or the object that contains it) to interact like a pawn:

Go *Details /Collision* and set "*Collision Preset*" to “*Pawn*”
In case you do not want to modify the whole collision preset, try to set only the “*Object Type*”.

11. Utilities Pawn Possession, Quality, FPS, Viewport Widget

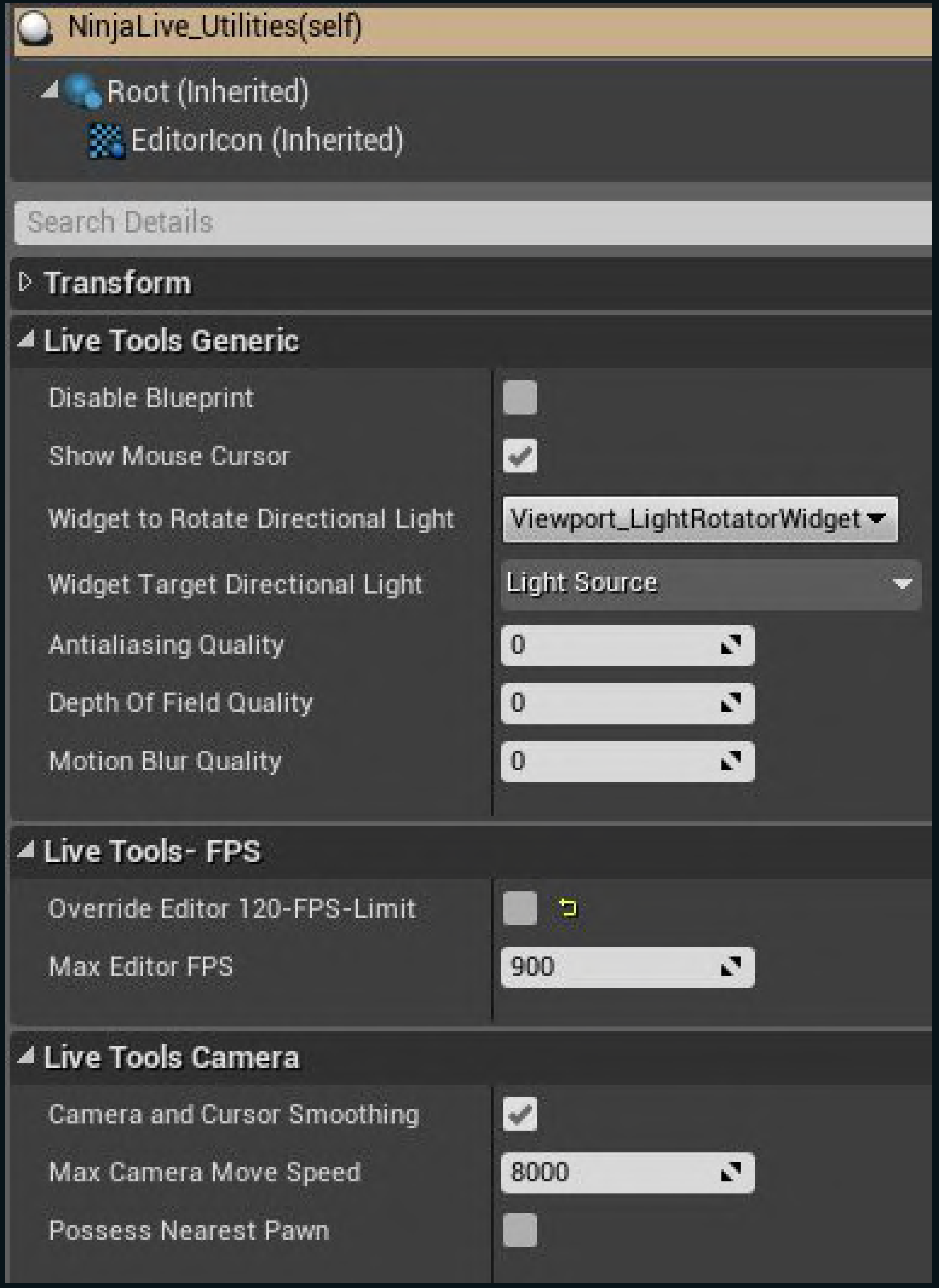
NinjaLive_Utilities Blueprint Actor could be optionally placed on game levels to help development. Symbol: a green letter N. Main features:

1. LIGHT WIDGET
   Add main (directional) light rotation and intensity controller widget to viewport
   To use this feat, you need to manually add the widget and the target light.

2. EDITOR FPS
   By default, UE is set to run at 120 FPS. As a simple kind of performance measurement, legacy ninja versions (until 1.3) were UNLOCKING the UE limit, using the following console command: *t.MaxFPS 900* *(900 FPS is an arbitrary, theoretical limit)*
   As a result, tutorial levels were running with no FPS-limit - usually around 200-400 FPS on a GTX 1070. This was a simple, good way to have instant feedback on performance bottlenecks. With the release of RTX 3000 series GeForce cards, a problem emerged: these cards were able to run test levels on 900+ FPS --- and the card got exhausted. The "Override Editor 120 FPS limit" got switched off, by default (on most levels).

3. POSSESSING PAWN
   Possess Nearest Pawn is a very useful function: In case it is OFF, we have a free / unbound spectator camera while *in Play*. In case it is ON, it will find the NEAREST pawn (compared to our current position in the world) and possess that. Ninja tutorial levels are often populated with multiple pawns at different stages - it is handy to navigate in editor, press Play and instantly possess that given pawn at that given stage.

4. The util (once placed on level) influences other SYSTEMIC things, like DOF, motion blur, antialias, camera and mouse-cursor smoothing.


NinjaLive_Utilities(self)
Root (Inherited)
EditorIcon (Inherited)
Search Details
Transform

Live Tools Generic

Disable Blueprint	☐
Show Mouse Cursor	✓
Widget to Rotate Directional Light	Viewport_LightRotatorWidget
Widget Target Directional Light	Light Source
Antialiasing Quality	0
Depth Of Field Quality	0
Motion Blur Quality	0

Live Tools- FPS

Override Editor 120-FPS-Limit	☐
Max Editor FPS	900

Live Tools Camera

Camera and Cursor Smoothing	✓
Max Camera Move Speed	8000
Possess Nearest Pawn	☐

14. Packaging, Compiling in general

14.1 Remove Preset Manager From Levels Before Compile

(A) Preset Manager is a developer feature: it is not needed for the final product to function
(B) PresetManager is using "Editor Only" functions (for file management) that could block compiling - it is advised to remove it from Game Levels completely once development is finished / before compiling

14.2 Please carefully study chapter 21.3.6 *Live Compatibility* in this document

Compatibility settings include *Trace Channel management* and *Dynamic Texture Lookup management* - both being critical / having an impact in project compiling / packaging.

(A) trace channel "autofind" does NOT work in packaged projects - users need to define trace and collision channels manually before packaging - this is a global setting (could be forced to all containers by defining trace channel variable values in the NinjaLiveComponent blueprint).

(B) dynamic texture lookup does NOT work in packaged projects

- METHOD1: define sim density and velocity input textures on a per container basis at *NinjaLiveComponent /LiveCompatibility /Overwrite*Input* (NOT a global setting)

- METHOD2: go to *Project settings /Packaging /Additional Non-Asset Directories to Package* and link the folder where the velocity and density maps are, this forces the system to package the maps. (GLOBAL setting)

14.3 NinjaLive 1.1 - 1.2 blueprints are branched to APEX and non-APEX versions

In case you are *not* using the APEX destruction plugin, you could completely remove all APEX related assets from the project before compiling. APEX assets are located at:
/Content /FluidNinjaLive /Versions
You could read more about the APEX removal in the Changelog: LINK (see v1.1 changes)
Important: in NinjaLive 1.3 and higher versions, all APEX related content is removed.

14.4 Mobile Rendering, Limitations

(a) Android graphics drivers are flipping RenderTargets vertically. Since Ninja is using a chain of RenderTargets, only ODD pieces (seem to) flip! To prepare the system for this, go to: *NinjaLiveComponent* Details /LiveCompatibility - and set "Flip RenderTargets for Mobile" to ON (before compiling).

(b) Parallax Mapping is not working on Android Mobile with ES 3.1.
Other features (like Raymarching) are tested: OK

(c) In NinjaLive 1.0 - 1.2 Tutorial levels contain Destructible Meshes to demonstrate the interaction of fracturing meshes / fluid dynamics / LOD. During mobile packaging, UE returns a warning: "DM_box01_SphericalMapped has a LOD section with 150 bones and the maximum supported number for feature level ES3_1 is 75." Since the referenced destructible is not used on mobile levels, you could ignore the warning - it does not block compiling / packaging.

14.3 RANDOM HINT: we could imitate mobile touchscreen in Unreal Editor, by:
Edit/Editor Preferences/Level Editor/Play/Play in Editor/Use mouse for touch: ON

15. Packaging to Android

Useful official guide from EPIC: Setting up for development for the Android platform
https://docs.unrealengine.com/en-US/SharingAndReleasing/Mobile/Android/GettingStarted/index.html

15.1 Prerequisites

(a) On the PC: install *CodeWorks* for Android
(b) On the Mobile: Go to /Settings /About device /Software Info /Build Number: tap 7 times to enable developer mode. In the settings root, search for "developer options", enable USB debugging.

15.2 setting up the Unreal Editor for Android packaging

go /Edit /Project Settings /Packaging /Build Config: Developing OR Shipping
go /Supported Platforms: set to Android
go /Edit /Project Settings /Platforms / Android: Enable APK packaging
go /Settings (cogwheel icon) /Preview Rendering Level: Android ES 3.1 (Shader (re)compiling!)

15.3 setting up Ninja for Android packaging

Android graphics drivers are flipping RenderTargets vertically. Since Ninja is using a chain of RenderTargets, only ODD pieces (seem to) flip! To prepare the system for this, go to: *NinjaLiveComponent* Details /LiveCompatibility - and set "Flip RenderTargets for Mobile" to ON (before compiling).

15.4 Packaging

go /File /PackageProject /Android /...(ASTC)
Select a Folder for Android package and auto-installer (Subfolder is automatically generated)

(a) You could follow the packaging process in the Output window. (b) Once done, plug your phone, go to the above provided folder, and start "Install*.bat" - (c) A console pops up with status, and in a minute the package is copied / installed on Android. (d) The package appears at the "All Apps" list with typic UE logo.

Setting up an Android Phone for developing / package transfer:
see UE documentation (enabling developer mode, allowing USB access at file transfer, finding the executable) + USB usage mode: charging VS USB for file transfer

17. Optimization

17.1 Use Activation Volume

By using the "Activation Volume feature", you could reduce the number of parallelly running sim containers.

Performance data suggests that we could use 1-2 high res area fx and 3-6 character fx simultaneously (on one screen) without taking over too much resources reserved for other game components. For an in-game cinematic the budget is more allowing: a single ultra HD (2k) - OR - 2-4 high res (1k) area fx - AND - 4-8 character effects.

Sim container sleep-wake state could be controlled at *NinjaLive Actor* Details panel /LiveActivation.

When placing containers, please keep in mind: Live is not a robust "whole world system", but a system supporting a multitude of dispersed, local interactions. See chapter 8 and 12 on performance.

17.2 Use LOD

Ninja is capable of distance-based quality reduction, regarding (a) sim sampling rate (FPS), (b) the number of pressure iteration cycles and (c) the number of tracked objects. See: *NinjaLiveComponent* Details /LivePerformance

17.3 Use Memory Pool Manager

Place a single RenderTarget Pool Manager actor on a (persistent) level and allow clients (*NinjaLiveComponent* owners) to acquire memory from the pool. The option to enable this feature is located at:
NinjaLiveComponent Details /LiveMemoryManagement /AutoConnectToMemoryPool

Advantages: (a) Generate a pool of RenderTargets in advance, (b) Let waking / sleeping ninja components [acquire from] / [release to] pool, (c) Avoid lag caused by gametime rendertarget creation and destruction, (d) Spare unreal garbage-collector load

17.4 Reduce the number of Trace Targets

NinjaLive is line tracing overlapping objects to project their position to the 2D sim plane. The number of trace targets matter - with 10-20 objects, we are fine. As soon as the number of targets exceeds 50, we could experience a 10-15% drop in performance. Optimize by reducing the number of tracked targets.

EXAMPLE1: DESTRUCTIBLES WITH MANY CHUNKS

Use the below option to make Ninja ignore a certain percentage (%) of chunks:
NinjaLiveComponent Details /LivePerformance /IngnoreDestructibleMEshChunks%

EXAMPLE2: SKELETONS

Mannequin skeleton is made of 68 bones. Usually, we do not need them all - eg: for a full body collision we do not need all the fingers. By manually providing selected bone names at the below input field, we could cover the full body:
NinjaLiveComponent Details /LiveInteraction /ContinuousInteractionBoneNamesExact

Tip: 14 Mannequin bones covering the full body (paste these list items to “BoneNamesExact” array)
head, neck_01, spine_01, spine_03, upperarm_twist_01_l, upperarm_twist_01_r, lowerarm_twist_01_l, lowerarm_twist_01_r, index_01_l, index_01_r, calf_l, calf_r, ball_l, ball_r

17.5 Disable Dynamic Asset Lookup (DAL)

Preset files could refer to bitmaps as density and velocity inputs. The preset stores only the bitmap's name and looks up the file dynamically, at sim initialization. This gives flexibility to the system, we could easily swap images and experiment. On the other hand, we need to handle this, to run the game in Standalone mode and to Package the game:

(A) OPTIMIZATION

Once done with experimenting and a preset file is "final", the dynamic parsing/lookup feature is not needed anymore - we could bypass it and "hardwire" a bitmap input to the blueprint, using the below options:

NinjaLiveComponent Details /LiveCompatibility /Overwrite Preset Density Input and /Overwrite Preset Velocity Input

The Issues and FAQ PDF contains a step-by-step guide on disabling DAL: *see i005*

(B) COMPATIBILITY

The velocity and density maps from the presets are dynamically loaded in real time but the package system of UE4 doesn't consider dynamically loaded assets as assets it should cook in the final package - so it doesn't include it.
In case users do not want to use the "overwrite method" (option “A”), go to...

Project settings /Packaging /Additional Non-Asset Directories to Package
...and link the folder where the velocity and density maps were *(/Content/FluidNinjaLive/Presets)*, this forces the system to package the maps.

17.6 Carefully plan Rendertarget resolution

RenderTarget resolution influences performance. RT resolution could be scaled globally and selectively at: *NinjaLiveComponent Details /LivePerformance*

17.6 Carefully choose the Pressure Solver (2 types) and solver settings

Simulation Pressure Field is calculated using an iterative process. More iterations result in more detail in turbulent structures / vorticity. More iterations also need more RenderTarget read/write operations per frame. Since RenderTarget write operations have a critical impact on performance, we are facing a TRADE OFF situation: performance VS structural details

Please have a look at this demo level: NinjaLive_Level02B_CriticalSettings - demonstrating the trade off. FIVE pressure iterations per frame (per render cycle) seems like a good compromise - 5 is set as default for NinjaLive. You could adjust this value at:
NinjaLiveComponent Details /LivePerformance

17.8 Reduce the number of "per Tick" refreshed Dynamic Material Params

To increase flexibility during VFX development, by default almost all fluidsim params are queried / refreshed / forwarded to sim materials ON A PER TICK BASIS.
Once your project is about to finish / you are done with param tweaking,
you could eliminate dynamic (per tick) param refreshing by putting non-changing values on a "do once" branch, or initializing them on "EventBeginPlay".
Params like Noise tiling and offset are typically targeted for this optimization: probably nobody is going to animate noise dynamically in game.

17.9 Blueprint Nativization (will be deprecated in UE5!)

Turn on *Blueprint Nativization* to improve the performance of COMPILED (packaged) project at *Project Settings /Packaging* AND in the Ninja Core Blueprints,
by opening the BP and visiting the *Blueprint Class Settings: Nativize = TRUE*

18. Changelog / Updates

This chapter is outsourced to a separate document: Link to PDF

19. Known Issues & FAQ

This chapter is outsourced to a separate document: Link to PDF

20. Niagara

Update: Live 1.7 has been released with Niagara two-way data flow:
(A) driving particles using fluidsim —- (B) driving fluidsim using particles

(A) is demonstrated on Tutorial Levels 20 A,B + UseCase Level 11
(B) is showcased on Tutorial Level 3 + UseCase Levels 12 A,B,C

20.1 Driving particle systems using real time fluid data

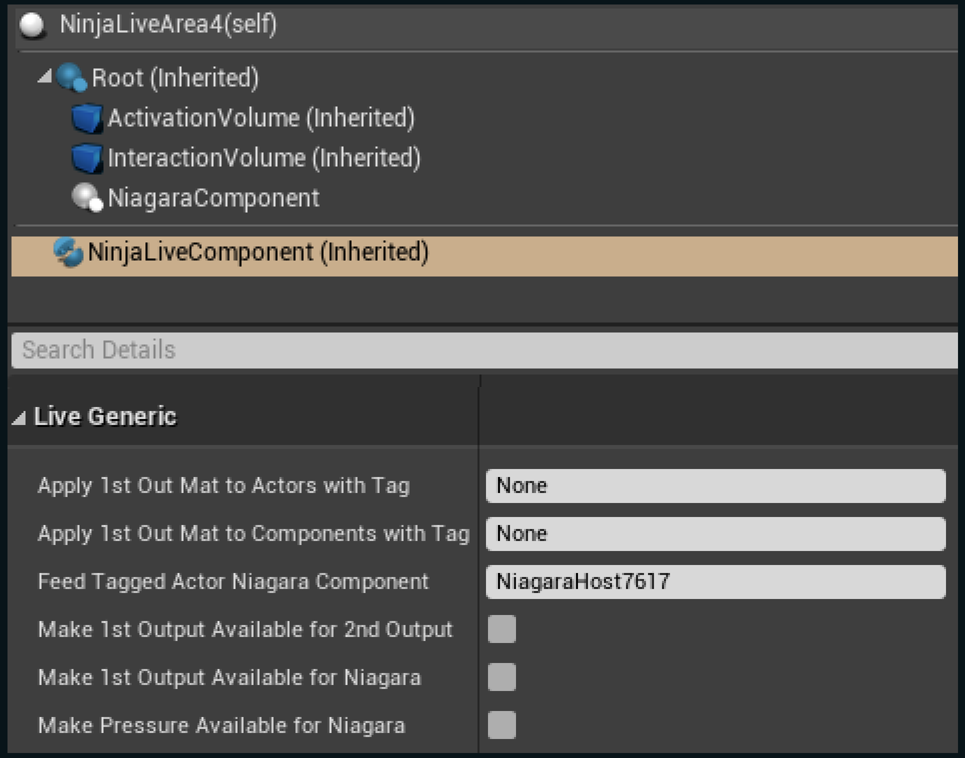
NinjaLive is performing real-time fluid simulation. The sim buffers could be used to drive Niagara GPU-particle masses. Eg.: Sim velocity field is ideal for accelerating particles, sim pressure is good for vertical-position-offset, density could be used to control spawning or mask opacity or for tone-mapping.
Niagara Systems on our example levels are using a 2D grid setup: particles are generated along an orthogonal grid and their XY position is used as UV to map them with 2D Fluidsim data, received from NinjaLive via the *TextureSample* Niagara Module.

See video demonstrating Level 20: LINK

20.3 LINKING NIAGARA AND NINJA

Update: Live 1.7 changed the way how niagara is driven (case A): feeding sim data to Niagara happens directly - no need to use on-disk RenderTargets anymore.
(See original guide at 20.3.B)

1. Tag the actor that embeds the targeted Niagara Component
   It could be any Actor - even the same actor that embeds NinjaLiveComponent
2. Provide Ninja with the tag, at:
   /NinjaLiveComponent /LiveGeneric /FeedTaggedActorNiagaraComponent



NinjaLiveArea4(self)
Root (Inherited)
ActivationVolume (Inherited)
InteractionVolume (Inherited)
NiagaraComponent
NinjaLiveComponent (Inherited)
Search Details
Live Generic

Apply 1st Out Mat to Actors with Tag	None
Apply 1st Out Mat to Components with Tag	None
Feed Tagged Actor Niagara Component	NiagaraHost7617
Make 1st Output Available for 2nd Output	☐
Make 1st Output Available for Niagara	☐
Make Pressure Available for Niagara	☐

20.3.B ORIGINAL STEP-BY-STEP GUIDE: LINKING NIAGARA AND NINJA STILL WORKING, BUT LIVE 1.7 RENDERED THIS OBSOLETE. SEE: 20.3

To do: we are writing ninja internal simulation buffer(s) to external RenderTargets and set a given NiagaraSystem to read these RenderTarget(s) as input

1. Create one or more RenderTargets in advance
   These are going to serve as a data-bridge to Niagara. Right click in the Content Browser, go to "Materials and Textures", choose "RenderTarget"

2. Set ninja to write the RenderTarget(s)
   At: NinjaLive Actor / NinjaLiveComponent /LiveGeneric
   a. switch on "Draw Internal RenderTargets to External"
   b. chose which buffers to export
   c. pick an existing (previously created) empty RenderTarget to write

3. Once the RenderTargets are created and ninja is set up to write them:
   Tell Niagara to read the RTs
   a. select a Niagara System on level that you would like to drive
   b. locate the actual uasset that is used (Details Panel /Niagara System asset)
   c. open the system with Niagara editor (double click on it in the Content Browser)
   d. locate the "Sample Texture" module in the Stack
   e. provide the pre-created RenderTarget as input in the "Texture" field

Ease of use: we could handle Niagara texture input as a parameter - so we can provide RenderTarget input on the *Actor Details* panel (no need to edit actual Niagara Systems when picking RenderTargets)

A. create a Texture-Sample type "User Parameter" in the Niagara System
B. provide this param as input to the "Sample Texture" module
C. set the value of this param by selecting the on-level niagara system, then going to the / Details Panel / Overrides

21. Complete List of NinjaLive UI parameters

21.1 Preset Manager: UI exposed params & functions
21.2 NinjaLive Actor params
21.3 NinjaLive Actor Component params

CHAPTER MOVED TO EXTERNAL PDF: LINK

22. Live fluidsim pipeline: technical description

CHAPTER MOVED TO EXTERNAL PDF: LINK

23. Using Sequencer to control NinjaLive Actors

Related example content: Level 5, Stage 4 (added to Live 1.0.0.4)
Related Tutorial Video: LINK

NinjaLive interactions and behaviour could be keyframed by *UE Sequencer*.
In general, we could animate (1) *effectors* - any agent that triggers NinjaLive response and (2) *sim parameters* - internal variables that change fluidsim behaviour and (3) *output material instance parameters*. Case "3" is not described in this document.

23.1 Effectors

An object / mesh could be effector if...

A. "*Generate Overlap Events*" option at *Actor Details /Collision* is flagged ON (by default, it is OFF) --- *See 10.5*
B. their "*Collision Object Type*" is set to one of the supported types (*WorldStatic, WorldDynamic, Pawn, Vehicle, Destructible, PhysicsBody*)
C. the class filters of the triggered (overlapped) container (NinjaLive Actor) are set to respond to the type of the given object. *See 10.1.6*

If an effector meets the above criteria and is keyframed to collide / overlap with a NinjaLive container it should trigger fluid response.
(Make sure you have properly set "Interaction Volume" size) --- *see 10.1.5, 21.2.2*

Note 1: Live generates fluidsim at runtime. To preview effector-fluid interaction, you should run the game (press play in editor). In case you are running Live in a Viewport, you could fit Sequencer to a nearby panel and edit keyframes / move the time-slider while the sim is running.
Note 2: One method to run the already keyframed sequence automatically at Play (and set the sequence to repeat) is to edit the *Level blueprint*. See Screenshot: LINK

23.2 Parameters

NinjaLiveComponent is using many params - most of them are set *only once*, at initialisation - these can not be animated/keyframed (typically system variables). A subset of params (typically fluidsim params) are being refreshed every tick - and could be efficiently animated. These params also appear on the Preset Manager UI. *See 21.1 for a complete list.*

Important: *NinjaLiveComponent* blueprint params are by default NOT accessible for the sequencer.

A. To make a given fluidsim param accessible for the sequencer: open */Content/FluidNinjaLive/NinjaLiveComponent* blueprint, look up the param in the "*Variables*" list, select it, go to the Details panel (in the blueprint editor) and flag *"Expose to Cinematics"* = ON. Compile and Save the blueprint.

B. To animate "Exposed" fluidsim params, add a given NinjaLive Actor to the sequencer Track list (+), then add NinjaLiveComponent as subtrack (+), then add the (preemptively) exposed param as subtrack (+)

23.3 Cinematics Rendering via MOVIE RENDER QUEUE (UE 4.26+)

NinjaLive is working fine with the new UE 4.26 Movie Render Queue
With the predecessor Render to Sequence it does not work!

A step-by-step guide

1. Edit /Plugins /MovieRenderQueue: ENABLE PLUGIN
2. Open the level that you would like to render
   (*NinjaLive_Level08_Demo_Roots* is ideal for testing: single container, no proximity activation)
3. Select "BP_NinjaLive_Utilities" on level and on the Details Panel:
   switch OFF the "Possess Nearest Pawn" flag
4. Content Browser: right click /CreateAdvancedAsset /Animation /LevelSequence
5. Place Actor Panel: drag a camera on level ---> set the camera to the needed position
6. Content Browser: double-click on the level sequence
7. In Sequencer: right click, "Actor to Sequencer"
   ---> add the recently placed Camera from the Level
8. Viewport, Top-Left corner UI /Perspective roll down menu: select Camera Actor
9. Window /Cinematics /MovieRenderQueue (MRQ)
10. In MRQ, click on the green "Render" roll down menu
    --> add the previously created Level Sequence
11. Adjust Output Folder and Config - if needed
    (by default: JPEG sequence, to "/Saved/MovieRenders/..")
12. Check 23.4: QUALITY NOTES
13. Press green "Render (Local)" in the Bottom-Right corner
14. After a few seconds of initialization, MRQ renders the image sequence to the provided folder

23.4 IMPORTANT NOTES ON QUALITY

Note 1: on many levels, sim containers are Proximity Activated (work only when the spectator/pawn is close) - and the cinematic camera does not necessarily trigger the proximity sensor. In this case, your sim container is passive. Disable proximity sensor at:
NinjaLive details panel /LiveActivation

Note 2: many times, distance based LOD is enabled on sim containers (lowers quality)
To make sure you render the sequence using the best available quality, disable LOD at:
NinjaLive details panel /NinjaLiveComponent /LivePerformance /LOD bool flags

Note 3: ninja is performing calculations in the 60 FPS range. When the Movie Render Queue is set to 30 FPS, it hurts ninja visual quality. Set MRQ rendering FPS to 60 and (if needed) achieve lower frame rates by skipping odd/even frames when compositing the output.

Related chapter in the Manual:
Chapter 23, Sequencer (LINK)

Related tutorial videos:
FluidNinja LIVE - Using SEQUENCER to animate Objects and Simulation Containers (LINK)
Improve Your Renders With Unreal Movie Render Queue (LINK)

24. Volumetrics

Example content: Levels 23-28, 30-31
Tutorial Videos: LINK1, LINK2 *+ Live 1.4 Volume Smoke quick preview:* LINK3

NinjaLive could drive 3D volumetric systems with real time 2D fluid sim data: users could set up responsive fog/smoke containers on level and paint dynamic cloud structures. Simulation density is used as a height-field, combined with 3D noise that is advected by sim velocity. Output is a true 3D volume, shaded by scene lighting. Comparing the three available systems:

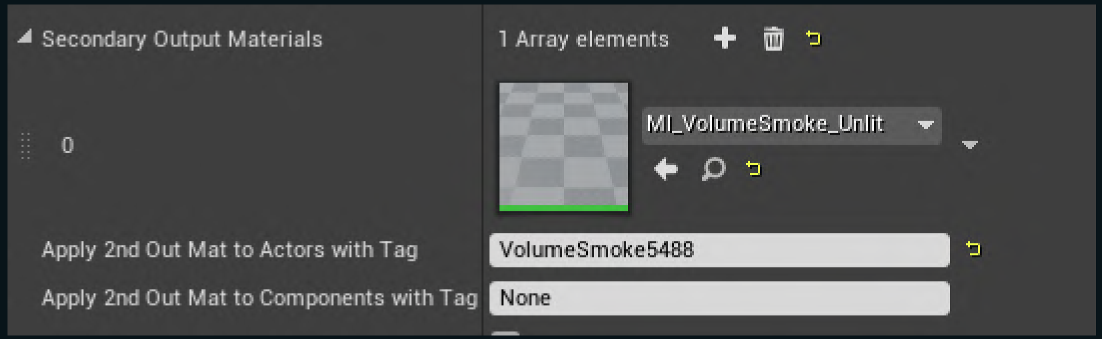
	FOG	SMOKE	CLOUD
Lowest supported UE version	4.23	4.23	4.26
UE native (built in) system	True	-	True
Material Domain	Volu	Volu	Volu
Volume bounds definition	Mesh	Mesh	UVW
Spatial usage	Local	Local	Global
Self shadows	-	True	True
Lit by directional light	-	True	True
Lit by point light	True	True	-
Receives Shadows	True	-	True

Update: in Live 1.7 a NEW VolumeSmokeComponent has been added, that could be embedded to any other Actor. The original (before Live 1.7) VolumeSmoke Actor has been rebuilt to CONTAIN VolumeSmokeComponent. New flags: "DirectlySet…" options enable VolumeSmoke to be controlled by ninja (see: 24.1, 29.1). "Anchor" forcibly keeps Vsmoke at a given location. 1<"VolumeSizeMultiplier" could EXTEND the volume beyond simulated areas and fill volume with 3D-Noise → see Tutorial Levels 30-B, 31-A,B

24.1 INTEGRATION

Update: Live 1.7 changed the way how volumes are driven: feeding sim data to Fog / Clouds / Smoke happens ***directly*** - no need to use on-disk RenderTargets anymore. *(See a detailed guide at 24.4)*

1. Add a Volumetric Material to Ninja.
   /NinjaLiveComponent /LiveGeneric /SecondaryOutputMaterials
2. Tag the actor that embeds the targeted Volume Component
   Note: NinjaLiveComponent and VolumeComponent could be in the same Actor
3. Provide Ninja with the tag, at:
   /NinjaLiveComponent /LiveGeneric /Apply2ndOutMatToActorsWithTag


Secondary Output Materials — 1 Array elements
0 — MI_VolumeSmoke_Unlit
Apply 2nd Out Mat to Actors with Tag: VolumeSmoke5488
Apply 2nd Out Mat to Components with Tag: None

Behind the stage, the material defined at pt1. is (a) being cloned to *DynamicMaterialInstance*, (b) provided with sim buffers via "TextureObject" params and (c) applied to tagged actors.

Warning: the three supported volume types (fog, smoke, cloud) are driven by three different BaseMaterial: */Volumetrics /BaseMaterials* – as a consequence, you can not drive "fog" with "smoke" type material, or drive "smoke" with a "cloud" material!

24.2 DEFINING VOLUME BOUNDS & POSITION

A. Volumetric Fog: a simple *StaticMesh Box* is placed on level, equipped with Volumetric Domain material- see *"VoluCube"* meshes listed in the *WorldOutliner /* placed *on Level*

B. Volumetric Clouds: bounds and position are defined within the material (no mesh needed), using UVW coordinate offset and scaling

- Select "VolumetricCloud" Actor on Level
- Go to Actor Details Panel, and double-click on "Cloud Material"
- In the Material, locate CloudU-Offset, *V-offset and *Altitude params

C. Volume Smoke: bounds are defined by a level-placed blueprint
Blueprint name: *VolumeSmokeContainer*

A debug box visualizes volume extension. To resize the volume, simply scale the level placed blueprint actor (use *uniform* scale).
Volume location is linked to the blueprint actor

24.3 TILING SIM SPACE: FULL SKY / GROUND COVERAGE

Live 1.2 contains a new feature: sim space *tiling* (by default OFF)
Usage and setup demonstrated on a new level: 24C

The feature enables us to compute only a small part of the sky (or ground-fog) - and use this part as a "tile" - the sim space is wrapped - so, we could cover the whole area (sky or ground) using repeated patterns.

Tiling could be set up / switched ON by tweaking the following params:

1. NinjaLiveComponent /LiveGeneric /SimAreaClamp bool flag - set to FALSE
2. PresetManager /SimAreaBorder settings - set all params to ZERO
3. VolumeFog and VolumeCloud material instances /Tiling bool flag - set to TRUE

More info: on setting up sim-space tiling: Level21 / Stage3
Tutorial videos on tiling - specific: LINK1 / generic: LINK2

24.4 STEP-BY-STEP GUIDE: LINKING VOLUMETRICS AND NINJA

Update: Live 1.7 changed the way how volumes are driven: feeding sim data to Fog / Clouds / Smoke happens *directly* - no need to use on-disk RenderTargets anymore.

- See 24.1, describing the new method
- Parts of the below text that describe "RenderTarget Management" and "how to apply materials on Volumes" are obsolete (still working, but not needed)

NinjaLive is performing real-time fluid simulation. The sim buffers could be used to drive volumetrics. For example: sim density could be used to define volume density, height and extinction. Sim velocity field is ideal for driving the flow of detail-noise.

Obsolete: we are writing ninja *internal* simulation buffer(s) to *external* RenderTargets and set a given volumetric material to read these RenderTarget(s) as input

1. Create one or more RGBA RenderTargets in advance, serving as a data-bridge to Volumetrics
2. Set ninja to write the RenderTarget(s) at: NinjaLive Actor / NinjaLiveComponent /LiveGeneric
   a. switch on "Draw Internal RenderTargets to External"
   b. chose which buffers to export
   c. pick an existing (previously created) empty RenderTarget to write

3. Place special UE Actors on level

   a. For *VolumeFog*: place a single Exponential Height Fog Actor on Level
      Set the "Volumetric Fog" option to ENABLED

   b. For *VolumeClouds*: place a single VolumetricCloud Actor on Level

   c. For *VolumeSmoke*: place VolumeSmokeContainer Actor blueprint on Level or add a VolumeSmokeComponent to a level placed Actor

4. About Volumetric Materials:

   a. In the case of *VolumeFog*, the volume-materials are applied on the level-placed StaticMeshes (see Level 23, browse "VoluCubes" in the WorldOutliner). Important: multiple meshes could be placed on level, and each mesh could be equipped with unique material. You could place your own mesh on level (any StaticMesh will do). Cuboids are practical + could be scaled non-proportionally.
      Do not forget to switch on "VolumeFog visualization" in the editor (Ctrl+F), in UE 4.26.2 it is switched off by default.

   b. In the case of *VolumeClouds*, the volume-material is directly provided in the VolumetricCloud Actor (no need to place a volume-mesh). You could set scale, offset and many other params in the material.

   c. In the case of *VolumeSmoke*, the used Volume Material Instance is defined on the NinjaLive_VolumeSmoke blueprint details panel. Depending on the param flags in the VolumeSmoke Material Instance, we should distinguish SIX material states (marked with No 1-6).

	UNLIT	POINT-LIT	DIRECTIONAL-LIT
VolumeNoise: FALSE	1	2	3
VolumeNoise: TRUE	4	5	6

Cases 1-4 are fast, ready for real-time gaming usage. Cases 5-6 are somewhat slower, mainly developed for Cinematic / NextGen usage.

24.5 ON DEMO LEVEL CONTENTS

Level 23	Volume Fog
Levels 24 ^{ABC}, 25, 26, 27, 28	Volume Clouds
Levels 30, 31	Volume Smoke

Spatial arrangement

On Levels 23, 26, 30 and 31 the sim containers (detecting collision) occupy the same location where the driven volume is. This setup enables actors to *seemingly* interact with the volumes. In fact, they interact with sim containers - and these drive volumes.

On levels 24A, 24B, 24C and 25 sim containers and volumes are *spatially detached:* (containers are running "somewhere else" / "behind the stages'') - this setup style is used when we do not need direct interaction between volumes and scene actors / we would like to "playback" a predetermined sequence of events - eg. a cloud vortex forming when summoning the final boss.
For “background” containers, we also need “non-camera based” line tracing - a new feature, introduced in Live 1.2 ---> explained in Chapter 10.2.6 + see Level 2C

Sim output display

In a classic NinjaLive setup, TraceMesh has two separate functions:

(1) It is capturing collision data	(intersecting with the line tracer)
(2) Simulation is mapped on it	(it is visualizing simulation output)

In volumetric setups, simulation output is rendered using the volume - so TraceMesh *loses* its role as a display (like on Levels 24-25-26)... or, it could be combined with the volume (both TraceMesh and Volume are displaying sim output) - like on Level 23.

Interaction

In a classic NinjaLive setup, TraceMesh captures collision input - and displays sim output at the same spatial location - we always see interacting objects and the simulation together. This setup allows us to conveniently *use the CAMERA as line tracing source*. Volumes could be easily detached from the sim container and display sim output "somewhere else": in these cases, the camera can *not* be used as line tracing source. Reason: when we see the volume, we don't necessarily see the sim container - eg. watching a skydome with clouds, while the container is "behind the scene".

To resolve the problem, NinjaLive 1.2 introduces *User Defined Line Trace Source*.
(1) UI:*NinjaLiveComponent Details Panel /LiveInteraction /UseCustomTraceSource*
(2) Dedicated chapter in this manual: 10.2.6 / Usage examples: on *Level 2C*
(3) Level 24-26 help texts explaining usage in situ (on the level)

Custom LineTrace Source combined with *disabled LOD* results in containers that could work “behind the stages” / away from the volumes - while not visible.

No interaction

There are cases when we don't need interaction at all: simulation input is a texture (static) or a material (animated), user interaction and overlap detection is *completely switched off* (at NinjaLive Actor Details panel /LiveInteraction), simulation is running without any interaction with the scene ----> See *Level 24B, 24C*

25. Detail Maps

A feature to add dynamic, flowing details to low and medium resolution simulations at minimal cost. Introduced to NinjaLive 1.3

We are using sim velocity buffer as a flow map to advect a user defined map (typically procedural noise) - and mix the dynamically flowing details to the native simulation output

Included examples:

- Level 29	adding cell noise to flame or combustion type fluids
- Level 11	adding a cloud noise to a smoke-sim
- Level 10B, Stage1	adding grainy "sand like" noise to viscous fluids

Accessible at: Output Material Instance /FlowMap parameter group

Concept behind Detail Maps:

Ninja fluid-sim pipeline is a collection of various data types like density, velocity, pressure, divergence. Density is generally handled like the "final product" while others considered as "by-products". How about utilising sim "by-products" for something useful? Pressure is ideal for optical and geometric distortion (see Level21 Stage6), velocity could be used to drive particles (see Level 20A,B) or: to drive texture advection - traditionally called "flow mapping". The flowmap concept: we are using two identical copies of a static texture and distort the image-pair using a shifted period oscillation. Distortion is driven by a velocity map. A velocity map is a field with directional vectors, ideal to tell particles / texels "which direction to go".

Ninja already utilizes flowmaps - this is how we make volumetric noise flow on clouds - see Levels 24-28. Now, an other flowmap feature has been implemented: detail mapping

26. Controlling Live in real time

To grasp the idea of *real time control,* load any ninja tutorial level, place a *Preset Manager*, start the Game *(Play)* and use Preset Manager to (A) interactively set Fluidsim Parameters, (B) change Input Textures and (C) Output Materials.
See videos: changing Sim Params: LINK / changing Output Materials: LINK1, LINK2

Similar control could be achieved by Game Logic (Blueprints, Code) or Sequencer.

- *Practical example1:* using NinjaLive Actor for an area FX. We would like to control the FX params using sequencer - to create a choreographed Cinematics Sequence
  See this video: LINK
- *Practical example2:* character ability FX is delivered by an embedded NinjaLiveComponent - we switch the fluidsim preset and the output material as the character uses various ability-FX

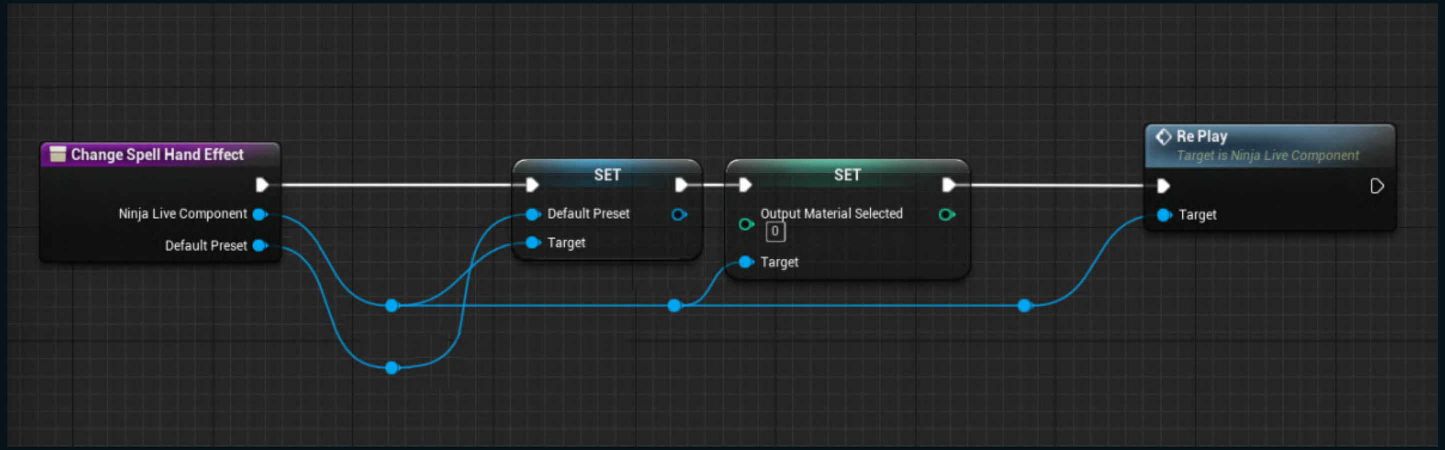
Control subjects in the NinjaLiveComponent blueprint could be grouped as:

1. Could be modified instantly
2. Could be modified by re-initializing NinjaLiveComponent blueprint

1. Simple Variables (eg. floats, integers) could be modified instantly by accessing NinjaLiveComponent. Variables have "telling names" - and you could read about them using the Tooltips and the Manual. Pls open up NinjaLiveComponent Blueprint: Preset Variables (for example) could be found under the "NONPUBLICLive Preset Variables" group. --- by code you can access them anytime. In the Blueprint that embeds NinjaLiveComponent (eg. a character) params could be directly accessed. For other (external) blueprints, the LIVE INTERFACE could be used. To modify a variable by Sequencer, you need to set "expose to cinematics" to TRUE. See Level 5, demonstrating how to use the Interface and Sequencer.

2. Asset-type-variables (like a DataTable with preset-values, or an OutputMaterial Instance) could be modified by re-initializing NinjaLive. (Trivial example: the Preset Manager re-initilizes ninja when changing materials or loading a new preset.)

---> See image below: Preset File and OutputMaterial index changing - combined with RePlay



Swapping Materials: only those output materials could be accessed (and used for swapping), that are previously added to the OutputMaterials array at LiveComponent /LiveGeneric
Swapping process: (1) changing the OutputMaterialSelected INDEX, then (2) re-initialize ninja

Please open NinjaLiveComponent Blueprint, and locate MODULE001. Check the *RePlay* node. As the label tells us: this event could be called remotely (eg. by an interface, or event dispatcher)

27. Mapping sim on 3D Mesh surfaces

See this video, demonstrating the usage of 3D meshes: LINK
The technique is UE5 compatible.

NinjaLive is a 2D fluid sim. Besides the original (A) camera facing flat planes and (B) parallax mapping, new techniques are being added to enhance and spatialize sim output.
Live 1.2 could utilize (C) UE Volumetric Fog and (D) UE Volumetric Clouds (see Chapter 24).
Live 1.4 introduces two more techniques:

1. raymarching based translucent volumetrics (entitled as "VolumeSmoke")
2. polygonal mesh based opaque 3D surfaces

This video demonstrates the two techniques: LINK + you could read about VolumeSmoke in Chapter 24. This chapter describes the mesh based solution.

Originally, Ninja used Parallax Occlusion Mapping (POM) to imitate depth. POM does not look good from a low camera angle. Please have a look at LEVEL 10-B. The fluidsim containers on this Level are all using OPAQUE Output Material, which creates a discrete surface - the fluid surface is NOT at all fog like (it is non-translucent) - but opaque, shiny, reflective - like blood, mud, snow.

The opaque surface property enables us to use real 3D meshes to display sim output:
by defining a pre-tessellated (hi-poly) grid as TraceMesh, and using simulation density as a height-field to drive vertex World Position Offset (WPO), we could dynamically distort the TraceMesh: the result looks good from low camera angles as well.

27.1 Reconfiguring a sim container from POM usage to 3D sim mesh usage

A STEP-BY-STEP GUIDE

1. Select Container
2. Select TraceMesh Actor Component
3. Replace "NinjaLiveTraceMesh" (a single quad plane) with a tessellated plane
   (eg. SM_plane_300x300)

4. Enable "Cast Shadow" at the mesh component Details Panel
5. Select NinjaLiveComponent
6. Go to LiveGeneric param group, Locate the currently used OutputMaterial in the array
   (checking the OutputMaterialSelected INDEX below the array helps)

7. Locate "Parallax" param group, disable ParallaxMapping
   (not needed when using a 3D sim mesh)

8. Locate "MeshDistort" param group, enable MeshDistortion, tweak params
   (pay attention to CLAMP!)

28. Moving in WorldSpace

Live 1.7 comes with great improvements in world-space movement.
Example content: Tutorial Levels 4, 32 + UseCase Levels 1, 2, 11

Ninja sim containers could be used two ways:
(1) Fixed position containers are here since Live 1.0: eg. puddle, bonfire, portal...
(2) Containers attached to moving Actors are improved through Live v1.2 - v1.7: imagine a large sim area attached to the player, always following, handling all interactions around the player - creating an illusion that we are moving in a full dynamic world.

28.1 When the sim container is travelling in WorldSpace, we adjust sim density and velocity according to global movement direction and speed. Methods:

A. Offset all simulation buffers inverse moving direction
Makes the already emitted fluid to behave inertly / lag behind the emitter
Key param: NinjaLiveComponent /LiveInteraction /TraceMeshMovingInWorldSpace

B. Add/subtract motion vector to sim velocity buffer
Push fluid towards - or against - moving direction
Key params: Preset Manager /Fields /Velocity Field Influenced by Sim Area Motion

Tweaking A/B could produce very different outcomes: eg. forcing the sim to lag behind, but also setting a negative velocity that pushes it along moving direction creates a fluid that creeps / lingers behind the emitter :)

28.2 Additional settings to tweak:

A. *Feedback* (Preset Manager UI /Sim section)
Defines "Trail Lifetime" - the time needed to fade out the lingering fluid

B. *Drag* (PresetManager UI /Brush section)
Stirring/dragging effect of moving objects on the simulation

C. Puncture-modify
NinjaLiveComponent Details /LiveBrushSettings /SimAreaMotionEffectsBrushPuncture
The Z component of sim area motion (perpendicular to TraceMesh) multiplies PUNCTURE --- in practice: when the sim container is moving towards/away from us, the "boiling" effect of puncture is more intense (by default) - and less intense when the param is set to ZERO

D. Latency
NinjaLiveComponent /LiveInteraction /LegacySimAreaMotionEffectsSimDensity

29. New features: Live 1.7

Interactive mini demo exe: LINK + Youtube playlist: LINK
Tutorial video covering this chapter: LINK

Live 1.7 is a major update. Features:

(1) driving other systems directly (no intermediate assets. eg RenderTargets)
(2) improved world-space motion
(3) local solver combined with global pattern generator → large responsive fields
(4) driving multiple systems with a single fluidsim (eg. surface + volume + particles)
(5) improved simple-mode: draw trajectories without running fluidsim (eg. footsteps)
(6) modularity: add others as Component to Ninja / add Ninja as Component to others
(7) niagara 2-way data flow: drive particles using fluidsim / drive fluidsim using particles
(8) tag based actor tracking

1. **Direct drive**
   Ninja could drive volumes, particles, foliage, landscape and water surfaces.
   Before 1.7, we used manually created intermediate assets to push data to other systems: RenderTargets for sim.buffers, ParamCollections for sim.pos & scale.
   In 1.7, we provide ninja with (1) a *material* that is going to be applied to the target system and (2) a *tag* to find targets → At init, Ninja is creating a Dynamic Instance of the provided material, assigning it to targets with all params and buffers directly set.
   To drive Niagara: we don't even need any material to set params.
   See 20.3 and 24.1 in this PDF for Niagara and Volume setups.
   See UE project: levels with "LIVE17" name-tag are updated to direct drive.
   Note: Materials and NiagaraSystems used in the project are all prepared to handle the incoming ninja data. Open up *NinjaLiveComponent Blueprint /MODULE023* to see the standard param names. Top 3: *VelocityDensityBuffer, TraceMeshPos, TraceMeshSize*

2. **Improved world-space motion**
   → see *Chapter 28*, explaining WorldSpace in general. Live 1.7 improvements:
   (a) Automatic WorldSpaceOffset / no param tweaking needed
   (b) WS-offset modes: users could choose between Normal, Manual, Quantized
   The quantizer divides WorldSpace to grid cells - and the local solver moves by "jumping cells" / moves in large discrete steps - instead of "every frame a bit". As a result, the number of texture-buffer interpolation ops is reduced: no data loss / blurring
   → demonstrated on *Tutorial Level 32*
   Mode selector: /NinjaLiveComponent /LiveInteraction /TraceMeshMovingInWorldSpace
   (c) Axis lock: by locking fluidsim area move on a given axis (eg. Z, vertical) we could easily set up surfaces where agents could "jump-off-and-back the sim" - ideal for large horizontal areas like water or sand.
   Key param: /NinjaLiveComponent /LiveInteraction /MovementIsLocked

3. **Covering large areas**
   Player-attached sim is generating fluid in a *finite* area. Sim data is seamlessly added to *infinite* noise and tile patterns: not obvious where sim area ends / non-sim area begins. Eg. local waves around moving bodies on a sea-surface, wheel-tracks in a desert.

   A NEW, GENERIC WORLD-SPACE BASE-MATERIAL IS INTRODUCED:
   /Content/... /OutputMaterials/BaseMaterials/M_NinjaOutput_WorldSpaceGeneric

   Instances of the Generic WS material are used in ALL new examples (sea, snow, water):
   → see *Tutorial Levels 23, 24-C, 30-B, 31-A,B* + *UseCase Levels 1, 2, 11, 13, 14-A,B,C*

   The tile-generator is implemented in ALL sytems: (a) in the *Generic WorldSpace mat.* and in *VolumeFog* material it is located under the TileMap Parameter Group.
   (b) In the *VolumeSmoke* and *VolumeClouds* mat: under Noise options.

4. **Driving multiple systems with a single fluidsim**
   Example: a single, pawn attached sim container is driving (a) water-surface,
   (b) volumetric vapor over the water and (c) Niagara splash particles

   Before Live 1.7, ninja had only a single OutputMaterial slot - and this slot
   was dedicated to TraceMesh:
   /NinjaLiveComponent /LiveGeneric /OutputMaterials

   Driving other systems happened indirectly, using manually configured on-disk
   RenderTargets, written by ninja, read by the target system.

   In Live 1.7 *tagged* systems could be directly driven via DynamicMaterialInstances.
   The FIRST OutputMaterial slot has been generalized: could be applied to any system.
   Plus a SECOND slot is added - also generic:
   /NinjaLiveComponent /LiveGeneric /SecondaryOutputMaterials
   Plus, Niagara also became directly accessible via tagging (no material needed)

   Sum: in Live 1.7 *three* systems could be driven *directly* + arbitrary number
   of systems *indirectly*, using the original "RenderTarget based" method.

   → see chapters 20.3 + 24.1 + 29.1 for DIRECT drive
   → see chapters 20.3.B.1-3 + 24.4.1-2 for INDIRECT DRIVE

   The two modes could be combined:
   → see level: *Usecase_015B_SandSimulated_PREVIEW_LIVE17*

5. **Improved Simple Painter Mode**
   Track objects and draw trajectories in world-space,
   without running fluidsim (ideal for footsteps, wheel-tracks)

   → see chapter 6.5: Simple Painter Mode
   → see Tutorial Levels 7, 32

6. **Modularity**
   NinjaLive Actor and NinjaLiveComponent are present since Live 1.0
   LiveComponent could be added to any user defined Actor.

   Utilizing the previously described, tag based *Direct Drive* (see 29.1), NinjaLive
   Component could access and drive OTHER COMPONENTS - eg. *NiagaraComponent* or
   VolumeCloudComponent. In Live 1.7, a *VolumeSmokeComponent* is also available!

   Target components could be under the same Actor as NinjaLiveComponent. This way,
   we could use NinjaLive ACTOR as a compact system, containing both LiveComponent
   and the driven components - eg. VolumeSmokeComponent and NiagaraComponent. We
   could find this kind of compact setup on new UseCase levels,
   for example on *UC 14-B,* by looking up *NinjaLive_Area_Water* Actor.

7. **Niagara two-way data flow**
→ see Chapter 20

(A) driving particles using fluidsim
Since Live 1.1, particles could be driven by fluidsim. Using the new WorldSpace, DirectDrive and Modularity functions, NiagaraSystems could be easily added to the same Actor where NinjaLiveComponent is hosted, and controlled using automated methods / without RenderTarget intermediates. Usecase 11 demonstrates 5 different ways to create fluid driven particle systems (Stage1: SimplePainter driven particles!)
→ see Tutorial Levels 20 A,B + UseCase Level 11

(B) driving fluidsim using particles
Live 1.7 has been released with UseCase 12 included: using the Niagara Modules of recently joined ninjadev Greg Resler aka Kynolin, we could capture particle data inside Niagara, and feed ninja with this. Particle dynamics is IDEAL to drive fluidsim!
→ see Tutorial Level 3 + UseCase Levels 12 A,B,C

8. **Tag based actor and component tracking**

Fact: NinjaLiveActor is handling interactions. Objects overlapping the *InteractionVolume* are identified, and the ID is forwarded to NinjaLiveComponent for ID based tracking.

Before Live 1.7 NinjaLive Actor overlap interactions could be filtered by CLASS (Collision Profile) definitions. For example: “detect Pawn type objects, ignore StaticMeshes ”.

In Live 1.7, TAGS could be used to FORCIBLY detect objects - independently from their Class and Collision Profile settings. See NinjaLive Actor Details Panel:

→ NinjaLive Actor /LiveInteraction /TrackActorPrimitiveComponentsWithTag
/TrackActorSkeletalMeshComponentsWithTag