FLUIDNINJA VFX TOOLS MANUAL

NinjaTools is for **baking** fluidsim to Flipbooks - and drive volumes, particle arrays and mesh surfaces using the baked data. For responsive, **non-baked** fluidsim: see **NinjaLIVE**

Latest NinjaTools version	1.8.54 for UE 5.4
Document updated	7 June 2024
Support:	andras.ketzer@gmail.com
Project homepage:	UE Marketplace
News:	Twitter
Tutorial videos	Youtube Playlist
Showcase videos	Youtube Playlist

Major updates

1.8 Volumetrics v2: support for UE 5.4 Heterogeneous Volumes
1.7 Deprecated Niagara Systems updated, Cascade legacy replaced with Niagara
1.5 VolumeFog and VolumeClouds support, Tiling
1.3 Driving Niagara
1.0 Baking, Looping

Contents

Intro CHAPTER 1

NinjaTools is designed to keep the process of VFX creation inside Unreal and make iteration quick and effortless. A set of tools integrated on a single GUI, baking real-time fluid simulation to animated textures, flowmaps, materials and vector fields.

In the core, there is a material based implementation of the Navier-Stokes fluid model that could use Niagara and Cascade particles and bitmaps as simulation input. The GUI is providing controls (sliders and numeric inputs) and a set of interactive viewports - displaying real-time in & output visuals. Params could be saved as Presets.
Baking is easy, looping + tiling is supported and the captured frames are played in a viewport besides the sim to make comparing and versioning easy. Following the baking process, a player-material is generated - embedding the baked data - providing params like playspeed, color, opacity, emissive, refraction…. etc

NINJATOOLS vs NINJALIVE

Both FluidNinja projects are focusing on fluid simulation inside UE. FluidNinja LIVE is for responsive, real time, interactive fluidsim. FluidNinja VFX TOOLS is for baked, non-responsive 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 (that could be used in the game) - while LIVE is a dynamic fluidsim (non-baked, runs together with the game), that could be placed in large numbers IN THE GAME, and it is creating real time interactive fluidsim. TOOLS and LIVE 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.

First run CHAPTER 2

Compiling shaders

Pre-compiled shaders are not included in EPIC store items and should be computed locally when you are loading the Ninja project for the first time in UE. Thank you for your patience, this could take ~2 minutes for the NinjaTools level and ~4 mins for the Content Examples level. When compiling is finished, start Ninja!

Engine Scalability Settings

Make sure you set Settings /EngineScalabilitySettings to “EPIC”
On lower quality, particles *will not be visible* in the ninja simulation-input viewport.

Viewports & Mouse Control

The suggested way to start Ninja is Play / *SelectedViewPort* or *NewEditorWindow*.
DO NOT start ninja using the *Simulate* option!
Make sure you are (1) going fullscreen first and (2) start PIE (play in editor) afterwards, otherwise the selectable screen area will be cropped. Another option to influence mouse behaviour: *Edit / Editor Preferences / Level Editor - Play / Game Gets Mouse Control.*

Help

Once Ninja is up and running: check the Intro module, a quick guide to the features of NinjaTools. The Status bar at the upper left corner is an Output log to push notifications on modding, saving and exporting assets. Tooltips are essential when you start experimenting with the NinjaGUI. Verbose explanation on all functions - activate it in the topmost GUI row.

Content

Ninja is primarily a tool - that comes with a large amount of sample assets to inspire users, and encourage modification / reverse engineering. Tools are located on the NinjaTools level, while sample assets are on the Content Examples level. Preset files, textures, materials and vector fields are located in the Input folder.

Connect Ninja to your project CHAPTER 3

Modes of using Ninja

1. As a standalone project - transferring only the baked flipbooks to your project
2. Including Flipbook Player materials to your project to fully utilize baked data.
3. Merging the full Ninja project to your project. Ninja is *300 MBytes* with Example Content and *60 Mbytes* without. The advantage of a full merge: NinjaTools exist on a separate Level in your project and could be used simply by switching to this level, without quitting your project. Plus the generated assets instantly appear in your project's Content folder. This is the recommended mode for using Ninja.

Include only Ninja Flipbooks to your Unreal Project (mode 1)

When capturing a simulation, Ninja is generating Atlas maps: multiple frames collapsed to a single image / single texture - as an array / tiles. Some call these Flipbooks, animated textures, image arrays. Unreal natively supports the playback of Atlas maps, via the SubUV and FlipBook functions - so they could be played without Ninja. Manuals on the topics:

https://wiki.unrealengine.com/SubUV_Particle_(Tutorial)
https://docs.unrealengine.com/en-US/Engine/Content/Tools/RenderToTextureTools/5/index.html

Still... we encourage you to use the FluidNinja custom player to access and play Atlas maps. Why? See the next chapter on *NinjaPlay*!

Using Ninja content in non-Unreal environment (mode 1b)

Atlas maps are used and supported everywhere. Which means, you can use Ninja to generate content for Unity, Blender and so on - by using the Export PNG option in the Sequence Player viewport, that pops up *during baking*.

Ninja is also capable of exporting image sequences - that could be used in *After Effects* or any other compositing software - and for *making GIF anims*. In the *FluidNinja* module, under the red coloured *Baking* options, change Rec.mode from *Atlas* to Sequence . Ninja exports the captured images as a sequence to a folder defined in the *NinjaConfig* module.

Include Ninja Flipbooks and Ninja Flipbook player materials to your project (mode 2)

You don't need to include the full Ninja project to achieve full player functionality. Once the assets are baked and container-materials are generated, you could migrate them to other projects. This means, you can use Ninja as a standalone tool, and use its output in other Unreal projects with all Ninja-features on. All you need to do is to migrate flipbooks (Texture 2D assets) and Flipbook players (Material functions, Materials and Material Instances).
See next chapter for details on *NinjaPlay / FlowPlay* and how to *migrate* them.

Merge full Ninja to your Unreal Project structure (mode 3)

Ninja, as a standalone project, is placed in the Content/FluidNinja folder. To include (embed / merge) it to other Unreal project:

A. quit Unreal
B. copy the downloaded *Content/FluidNinja* folder to your project's *Content* folder
C. start Unreal and load your project.
D. go to the following menu: Edit/Plugins/Scripting/ and enable the standard Unreal Editor Scripting Utilities plugin
E. save your project and restart Unreal
F. once restarted and your project loaded, go to Content/FluidNinja/Levels and *double-click* on the *NinjaTools* level (load the level)
G. once loaded, go to the World Settings window and in the Game Mode submenu set the Game Mode Override from "None" to "BP_NinjaGamemode". Reload the level.
H. Make sure that Settings /EngineScalabilitySettings is set to "EPIC"
I. NinjaTools 1.8 comes with included PAWN examples. This PDF describes how to set up pawn control in a merged ninja project: LINK

NinjaPlay CHAPTER 4

NinjaPlay is an Atlas map player / Flipbook player, implemented as a material. There are two versions - Basic (mostly for previewing atlas maps) and Advanced (for production). You can instantly generate NinjaPlay *container materials* for your baked simulations by using the Materials option in the Sequence Player viewport that pops up *during baking*.

1. NinjaPlay utilizes three frame-blending modes: (video link)
    a. no frame blending: quick and efficient (GPU-wise) method, optimal when playing frames using a high FPS (play speed value > 2)
    b. linear frame blending between frames / simple fade from a frame to the next
    c. velocity based frame blending (aka. vector space interpolation): unique Ninja feature, using velocity data (extracted from the fluid simulation) to figure out the “traveling direction” of pixels and move them between source and target positions (the frames of a sequence). Optimal when playing a sequence on low speed (slow-motion), results a smooth looking motion using only a few frames - eg. a realistic, dense smoke-column loop in 16 frames

2. NinjaPlay provides dozens of params to tweak baked sequences (after quitting Ninja) in the actual Unreal scene where the baked data is used - color gradients, emissive power, opacity, matte, play speed and additional feats like geometry distortion and refraction. Params could be set *manually* on the NinjaPlay *Material Instance* Details panel, or *remotely / indirectly* by Blueprints and Particle Systems.

3. There is a dedicated parameter to control time: CustomTime - using this channel, NinjaPlay could be forced re-time the played sequence based on external params (eg. a *Timeline* node in a Blueprint, or *Dynamic* graph in a particle system). The result is a “bullet time” like FX, baked sequences could be suddenly slowed down or fast forwarded (combined with the above mentioned velocity interpolation)

Dedicated document on NinjaPlay: link
Videos: features of NinjaPlay Container Material: link / Baked data on Cascade particles: link

Migrating NinjaPlay

Both Basic and Advanced NinjaPlay BaseMaterials are located here:
FluidNinja/Core/AtlasPlayer/…
M_AtlasPlayBasic
M_AtlasPlayAdvanced

Both materials are using the same three material functions for frame blending modes:
FluidNinja/Core/AtlasPlayer/…
MF_AtlasPlayFunction1 , 2 , 3

Both *M_AtlasPlayBasic* and *M_AtlasPlayAdvanced* are utilized through Material Instances. We do not modify our *BaseMaterials* or apply them directly on surfaces. We are manually creating or automatically generating instances of them and working with these instances. Each instance could hold unique control parameters and unique input textures - an efficient way to create variations while keeping memory usage low.

To migrate NinjaPlay (AtlasPlay) as a functionality to another Unreal project, you need to copy the above mentioned 3 *Material Functions* and 2 *BaseMaterials* to your project, and re-link them (tell the BaseMaterials where the migrated functions are located).
Note: you need to do this only once (per project). All incoming *Material Instances* are going to use the same BaseMaterial(s).

To migrate a specific baked sequence and the belonging container material-instance: copy both the Texture assets and Material Instances to *your* project. The migrated Material Instances should be re-linked (pointed) to the already (preemptively) migrated BaseMaterials (Details panel /General /Parent). Input Texture assets should be re-linked as well. Otherwise the migrated Material Instances remember their settings - except the above mentioned Parent material and Texture input.

FlowPlay CHAPTER 5

FlowPlay is a modified version of NinjaPlay, optimized to distort an arbitrary target image in real-time using baked velocity frame(s). Complete tutorial: video link

Basic use: Flow is a low cost (GPU load, memory) way to create VFX. With just a single velocity frame and a single flow target frame, a swirling, animated visuals could be created. See the Content Examples level to see how these work, or check this video insert: link

Advanced use: Flow could be imagined as an extra layer on top of NinjaPlay. The fluid simulation output could be pushed to a higher level of complexity by adding an extra map that *flows together* with the animated map, while adding extra details to it. Examples: (1) imagine a smooth, low resolution animated smoke as Ninja output, combined with a high resolution noise map that adds fractal details to the smoke - (2) or imagine a flame anim using only 8 frames, overlayed with noise that flows turbulently and smoothly - resulting a realistic high frequency flame behaviour. See the (1) Content Examples level to see how these work, or (2) the next page in this doc, or (3) check this video insert: link

You can instantly generate FlowPlay *container materials* using the Output / Materials option in the NinjaFlow program module. The module is meant to link velocity and flow target maps together and save them as a material - as the first step of VFX development. The preview is very crude in the module (no transparency, no velocity based frame blending, no coloring). The major part of flow visual dev. comes after quitting Ninja - via tweaking the exported FlowPlayer material on the Material Instance *Details* panel.

FlowPlayer has four versions - Basic / Advanced x Single Frame / Multi Frame. Depending on the number of velocity input frames (single vs multiple) the NinjaFlow module automatically decides to save a single / multi frame material.

Migrating FlowPlay

Basic / Advanced x Single / Multiframe FlowPlay BaseMaterials are located here:

FluidNinja/Core/FlowPlayer/…

- M_FlowPlayBasic_SingleFrame
- M_FlowPlayBasic_MultiFrame
- M_FlowPlayAdvanced_SingleFrame
- M_FlowPlayAdvanced_MultiFrame

All materials are using these material functions for frame blending and flow distortion:

FluidNinja/Core/AtlasPlayer/…

- MF_AtlasPlayFunction2
- MF_AtlasPlayFunction3

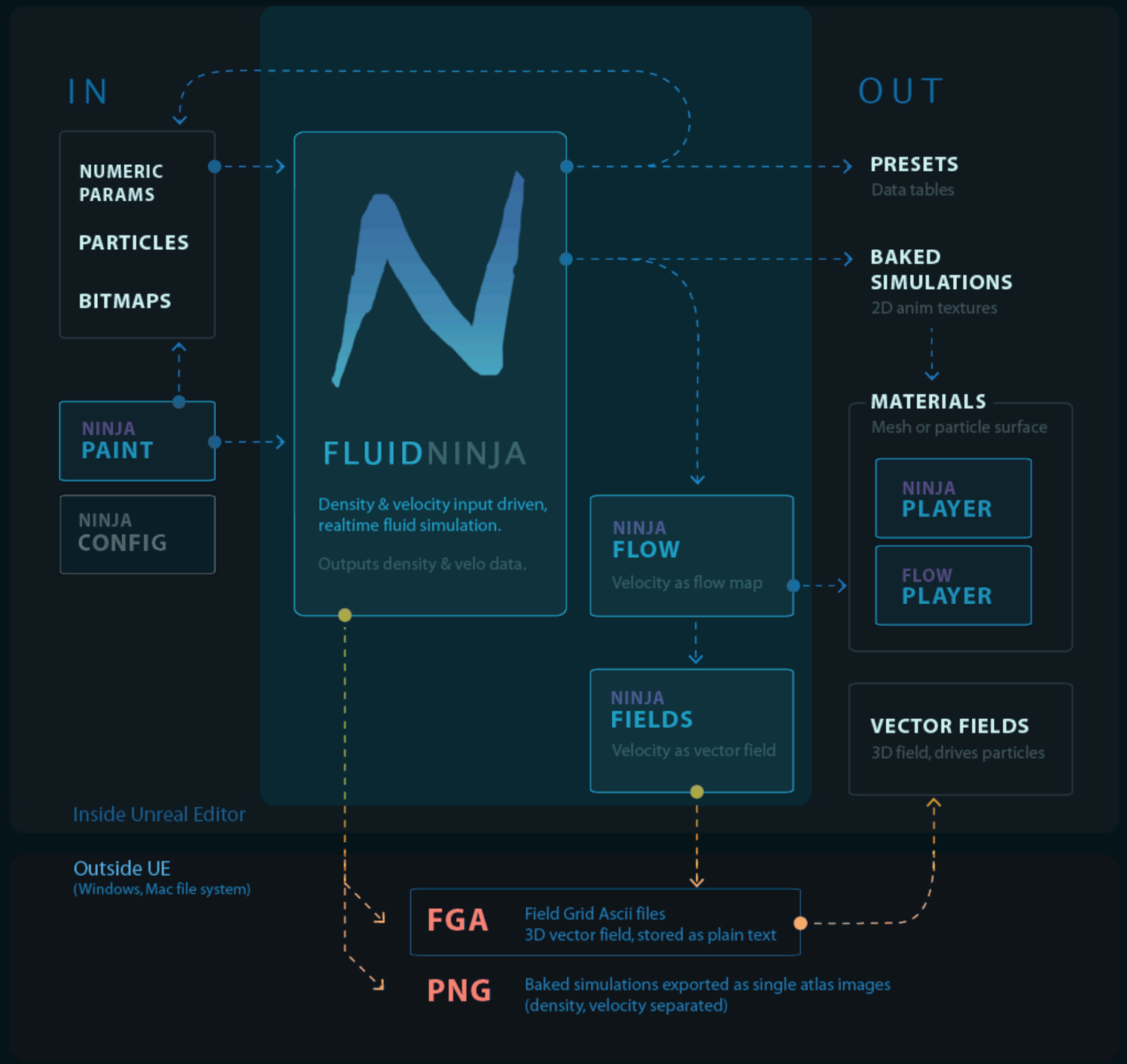
FluidNinja/Core/FlowPlayer/…

- MF_FlowMapFunction

All materials are utilized through Material Instances. We do not modify our *BaseMaterials* or apply them directly on surfaces. We are are using instances of them and work with these.

To migrate FlowPlay as a functionality to an other Unreal project, you need to copy the above mentioned 3 *Material Functions* and 4 *BaseMaterials* to your project, and re-link them (tell the BaseMaterials where the migrated functions are located).
Note: you need to do this only once (per project). All incoming *Material Instances* are going to use the same BaseMaterial(s).

To migrate a baked sequence + flow target texture + the belonging container material-instance: copy both the Texture assets and Material Instances to *your* project. The migrated Material Instances should be re-linked (pointed) to the already (preemptively) migrated BaseMaterials (Details panel /General /Parent). Input Texture assets should be re-linked as well. Otherwise the migrated Material Instances remember their settings - except the above mentioned Parent material and Texture input.



Ninja Modules CHAPTER 6

The FluidNinja project is set of tools (NinjaTools) integrated on a single GUI. The functional units of NinjaTools are called modules. Modules are linked together by data flow.

Intro	quick guide to the features of NinjaTools
NinjaConfig	asset and color management
FluidNinjaCore	create density and velocity maps based on particles, textures or NinjaPaint input
NinjaPaint	paint simulation input in real time
NinjaFlow	use velocity maps to animate any texture
NinjaFields	turn velocity maps to vector field data

Note: FluidNinja output data is feeding the Flow and Field modules. The freshly baked simulation data should be "Marked for save" in order to be instantly available as Flow/Field input.

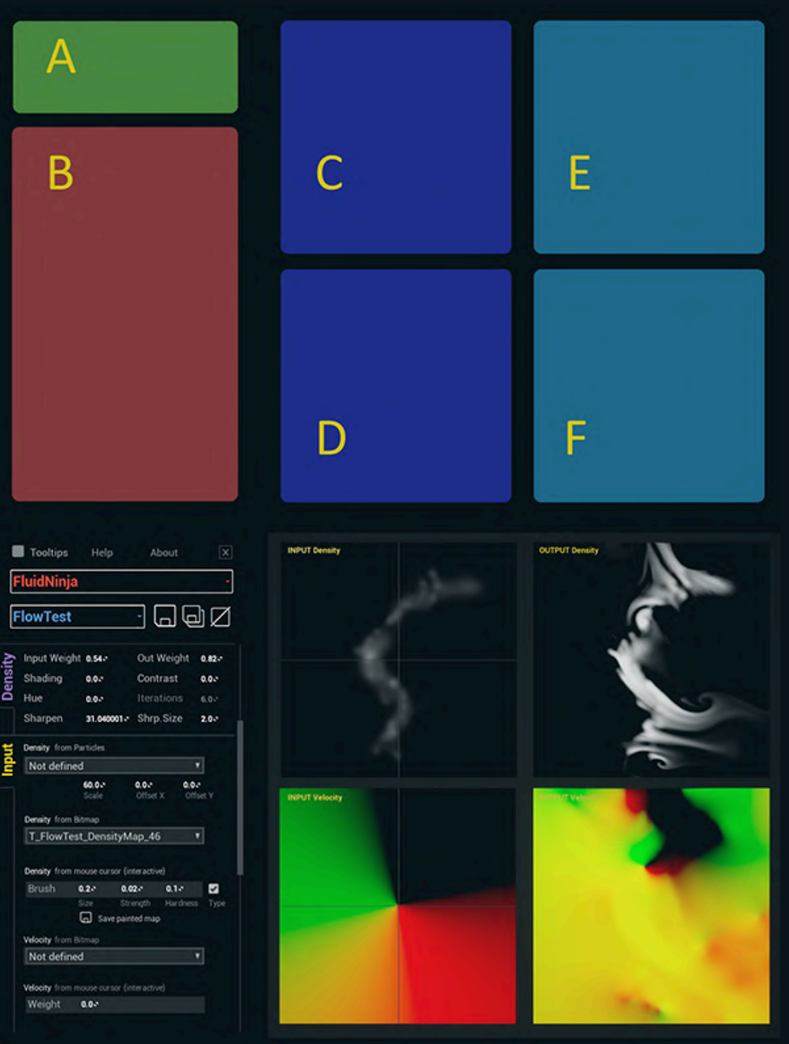
Introduction to the FluidNinja main module: video link
Quick intro to NinjaConfig, NinjaFields and NinjaFlow: video link
Using NinjaFields, in details: video link1, video link2
Using NinjaFlow, in details: video link
NinjaPaint: video link1, video link2

Niagara Modules added in Ninja v1.3

The modules are utilized in Niagara Editor (“inserted in the stack”). Modules could sample FluidNinja Core output data, so we could drive Niagara Systems with fluid data. See three Niagara use case levels, tutorial videos and a dedicated Niagara manual:
drive.google.com/file/d/19FAnIfwJHfMHNUnkEApoVu18LHgPjVN4

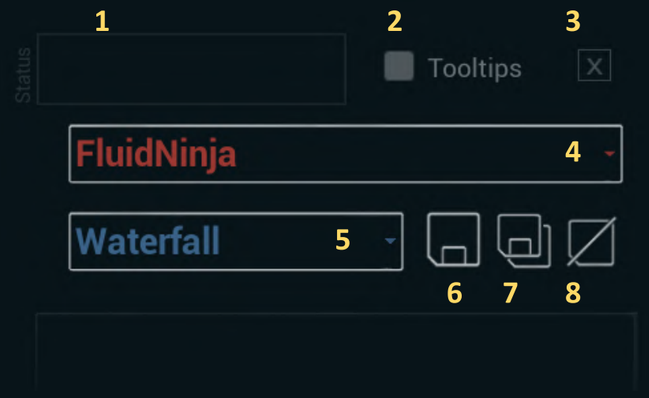
GUI CHAPTER 7

A.	GUI header	navigation and data management
B.	Sidebar	module specific operations
C. D.	Input viewports	realtime input density, velocity
E. F.	Output viewports	realtime output density, velocity



GUI header

1. Status — notifications on modding/saving/exporting assets
2. Tooltips — provides help for UI functions as Tooltip
3. Quit Ninja
4. Module selector



5. Preset selector
   You can browse the preset list via keyboard up/down arrows

6. Save preset (overwrite)
   Instantly updates the original preset with current settings

7. Duplicate preset (save as)
   Writes current settings to a new file and asks for a file name

8. Remove preset
   Instantly removes preset from list
   The file is preserved as: DT_RemovedPreset_[original preset name]

   Note: before closing Unreal Editor, saved and duplicated preset files must be written to disk permanently via "File / Save All"

Baking CHAPTER 8

When you start baking in FluidNinja module (by pressing the red, circle shaped Record button [1]) two new viewports pop up [2, 3], temporarily overlaying the original viewports.

The Baked Sequence viewport [3] is displaying a playback of captured frames - so you can preview your baking test and compare it with the realtime sim running above. Check looping and composition: repeat the baking process (simply by pressing REC again) to produce versions, save only the best.

The Sequence Player viewport [2] is providing tools to manage the baked data. Set playback speed and frame blending type, save the baked data to a Flipbook (Texture Uassets) or export as PNG, generate container materials (NinjaPlay / FlowPlay) that embed and play the Flipbook plus provide params for later tweaking.

Have a look at the tutorial video on baking: video link

When you quit Ninja, simply drag-n-drop the freshly generated container materials to your in-scene objects, and start tweaking color, emissive power, opacity, matte, play speed.

Presets CHAPTER 9

Parameters for Ninja modules are loaded / saved as a preset. Presets are stored one-by-one as *DataTable* uassets. Presets could reference other assets (textures, particle systems) and pull them as input for simulation. Asset references are called Templates.
Note: Ninja*Flow* and Ninja*Fields* could pull *any asset* under the NinjaRoot folder domain (Content/FluidNinja), while the *FluidNinja module presets* have a limited scope to avoid cross-referencing. Check the tutorial on Preset Scope: video link

1. Normally, the preset contains only the name of the referenced asset (eg.: T_noise), and the preset loader looks for the template in the same folder where the given preset file is located. Keep presets and their belongings together! (Preset Packages)

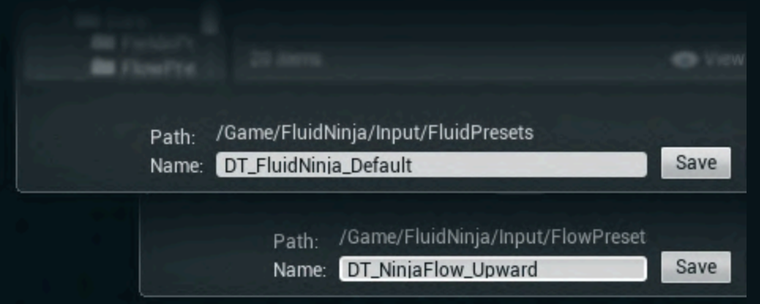
2. Templates could be referenced by absolute paths (eg.: /Game/Textures/T_noise), by manually pasting the path to the preset *DataTable* file in UE Content Browser. Not recommended: absolute paths work only in your folder structure and (a.) fail at other users if the preset is shared, (b.) fail locally if you modify your folder structure, (c.) modding a shared asset will change the visual style of all dependent presets.

It is possible to export a preset as CSV by right clicking on it in Content Browser - also, CSV files can be imported as DataTables. Take advantage of this portability: we encourage you to share presets and preset packages with other FluidNinja users!

Important: presets are recognized only when properly named. The naming convention is to separate_components_ with_underscore, plus put a *DT prefix* and a *Module name* before the actual preset name. Ninja is using the ModuleName part to match presets with their parent modules. Eg.: *DT_NinjaFlow_Turbulent8, DT_NinjaFields_Orb*

In case of *FluidNinja* presets, placement influences the scope of available input assets.
A preset is looking for input assets in its own folder and its subfolders: higher preset placement in the folder hierarchy = more available input assets. In general, it is best to keep presets in separate subfolders with their own templates to avoid cross referencing. Check the *PresetPlacementExample1,2* presets in the FluidNinja module to see how it works.

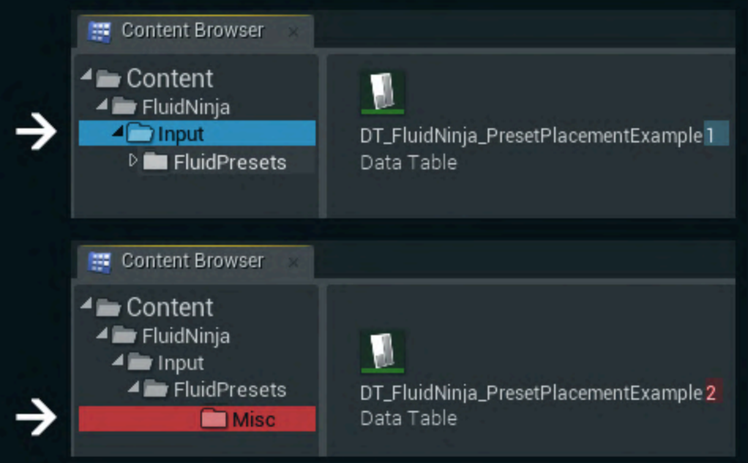
Preset naming



Example:

DT_	FLUIDNINJA_	PYRO2
PREFIX	MODULENAME	CUSTOM PRESETNAME
fixed		arbitrary

Preset placement = Scope



Higher preset placement in folder hierarchy = more available input assets (bitmaps, particles)

FluidNinja module CHAPTER 10

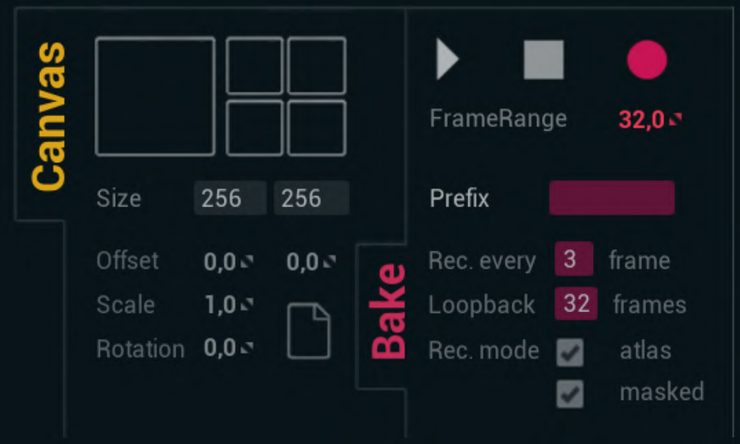
The main module of FluidNinja VFX Tools, performs velocity and density input driven real time fluid simulation, capable of baking density and velocity flipbooks, materials and vector field data. Let's skim through the sidebar options! Tutorial: video link
Note: when running Ninja, the tooltips provide you with accurate, detailed information on how-to-use a feature.

Canvas

Set render canvas properties: size, offset, scale, rotation.
Switch between Single fullscreen / Quad viewports.
Clear / erase buffers (canvas) during simulation.
Recording multiple frames: the output image resolution comes from Canvas Size X,Y and FrameRange. Eg.: canvas size is 256x256, FrameRange = 16 ---> The frames are recorded in a 4x4 matrix with a resolution of 1024x1024

Bake

Control sim with Play, Pause, start baking with Rec.
PlaySpeed could be adjusted with +/- by enabling *CustomTickRate* in NinjaConfig.
FrameRange: the number of frames to record
Prefix: define the name of recorded atlas images
+ auto naming / numbering when no name provided
Rec. every N-th frame: set the distance between recorded frames. 2-5 is ideal for smooth feel. Above 12, the recorded seq. starts to look like a stop-motion animation
Loopback: crossfade the ending part of sequence to the start, to create a smooth looping anim
Rec.mode: atlas vs image sequence
Masked: mask velocity with density



Notes on Recording / Baking

The system starts capturing frame(s) based on current *Size* (image resolution) and *FrameRange* settings. By default, the frames are being recorded like "tiles" on a single composite image (an Atlas map): eg.: 16 frames form a 4x4 tile set.

1. Ninja Simulations are Non-deterministic / influenced by rnd factors. Record many temp versions, take your time with experimenting.

2. When the recording starts, a new panel pops up - to preview and manage the recorded sequence (see the Pop-up GUI chapter).
    a. Sequences are recorded to memory and automatically erased when you quit unreal. To keep one, you have to "Mark for save". This feat enables you to record many temp sequences, and mark only the best few
    b. It is recommended to save a Container material for your atlas images. This is a special material that plays the baked flipbook once you quit Ninja and could be drag-n-dropped to meshes + provides many parameters to finalize the visuals

3. A trick to record non-looped sequences (eg. an explosion)
    a. set the particle input to repeat the non-looped input
    b. stop the simulation when a cycle ended and before a new starts
    c. set up all the parameters, and press record: the baking will not start until you press play (since baking is dependent on frame rendering)

Velocity

Offset the velocity field to give it a uniform direction
Rotate the velo field to create a circular flow
Amplify field strength, negative values invert direction
Turbulence causes more swirling
Noise is masking the strength of input velocity field

Density

Input weight controls the virtual density of input data in the simulation. More density causes bigger pressure, faster outflow, more fluctuation. Areas with lower density values are gradually dissolving, only the max density value [1] preserves a density spot from being dissipated. If you are using a brush input (paint), and would like to keep the painted shape on canvas (make it a persistent source of density) set weight to 1 - otherwise, it is fading out.

Output weight influences the speed of dissolve (the "lifetime" of a virtual density particle)

Shading fakes directional light on the simulation, to give a bit of depth to smokes, emphasize edges on flames

Contrast: cartoony look could be achieved
Hue: colorize density data. Important: use this to test/preview only! It is advised to bake density maps in grayscale, and colorize them
later using the container materials (NinjaPlay, FlowPlay)
Iterations: lower values could improve visual performance when working with extreme canvas size (eg. calculating frames in 2k resolution)
Sharpen. Trick: sub-zero values cause blur!
Shrp.size influences frequency domain

Input: Particle Systems

FluidNinja could use Niagara and Cascade particles, bitmaps and real time painting as density input for the fluid simulation - and procedural or bitmap based input for velocity. To set input type, visit the Input tab on the sidebar. Tutorial on the topic: video link

Density from Particles

In this roll down menu, you could pick Niagara or Cascade particle systems to serve as density templates for your Ninja preset. Using the Scale and Offset XY params below the input field, you can zoom on particle emitters and reposition them in the simulation area. The position of an emitter (density source) relative to the input velocity field is very important: the direction of flow could change simply by moving the emitter on the canvas (if the velocity field is not homogen).

Note 1 : only those particle templates are listed / visible that are located in the same folder as the preset that calls them. (eg. /Game/FluidNinja/Input/FluidPresets/Pyro). Subfolders are also listed, but Ninja is going to reference these with an absolute path. Absolute paths are indicated with a yellow triangle warning sign besides the input field. You could feed your preset with particles by moving the particle uassets to the folder where your preset is - or placing the preset file higher in the folder hierarchy to broaden its scope to see more input assets. It is best to keep presets and their templates together (in the same folder) to avoid cross referencing. Advanced: you could manually edit the Data Table of a given preset in Unreal Editor to insert reference to any template. Copy-paste its absolute path into the "Template" field of the Data Table. (eg. /Game/FluidNinja/Input/FluidPresets/Pyro/PS_Pyro7). Tutorial on absolute paths: video link

Note 2 : Sometimes, particle systems look slightly different in (1) Particle Editor (Cascade or Niagara) AND (2) Ninja Density input viewport. To sync visuals, to following check-ups are suggested:

A. Switch particle systems type to LOCAL in the Particle Editor (as opposed to GLOBAL).
[In Niagara: Emitter Properties / Local Space: ON] [In Cascade: Emitter / Required / Use Local Space: ON]

B. Try closing the Niagara/Cascade editor while running Ninja (the two, simultaneously running GPU sim might interfere - the one in Ninja and the same in Particle Editor)

C. Ninja is capturing the particle simulation using a fixed view angle, that might be different from your view angle in Niagara/Cascade editor. In case if something does not look the same, try to swap/flip axes (eg. something is aligned on the Z axis - and change that to Y).

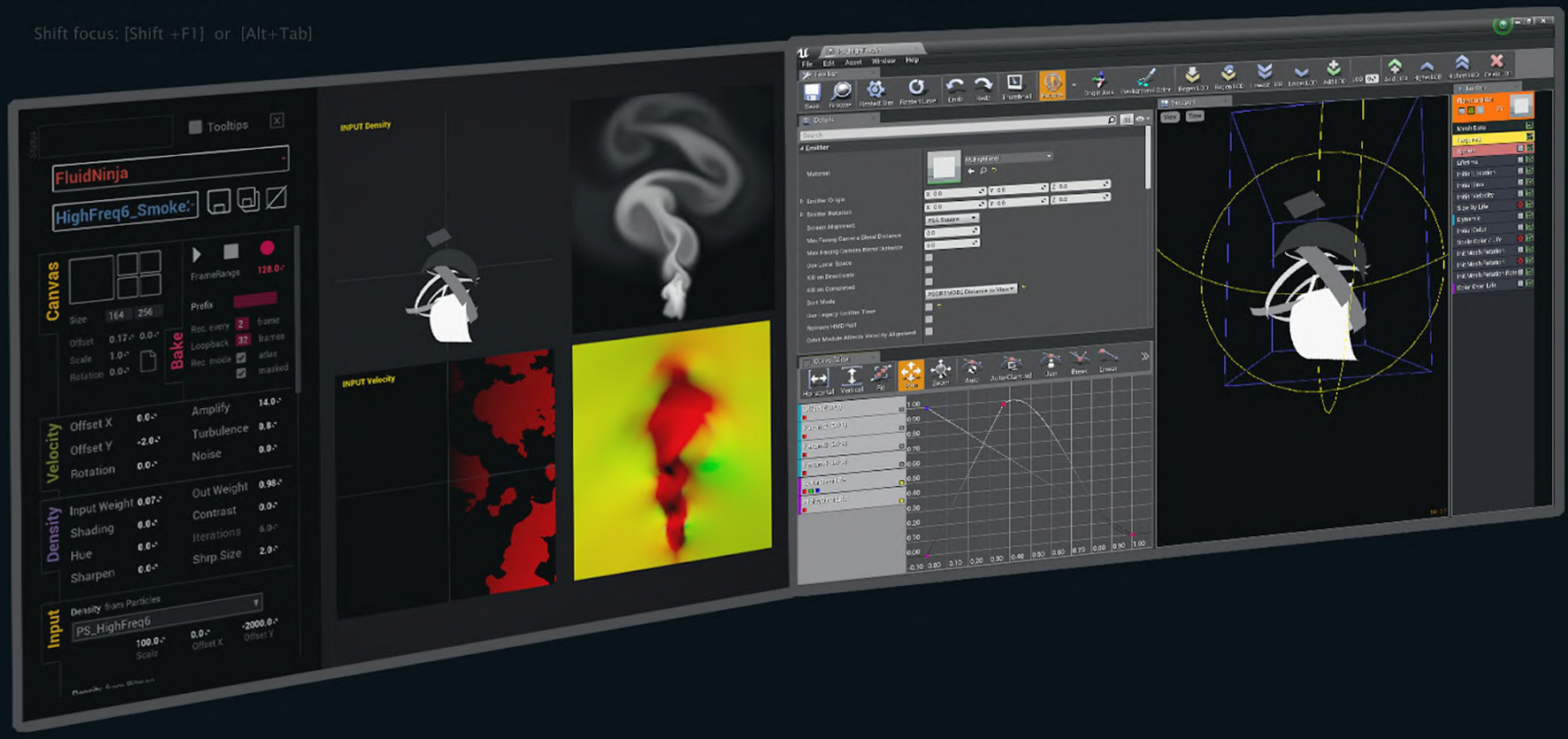
SCREEN1: **NINJA**

Pick a particle system as density input

Shift focus: [Shift +F1] or [Alt+Tab]

SCREEN2: **CASCADE**

Edit the same particle system in Cascade.



FluidNinja / Particle Editor dual screen setup

An efficient method for working with a Particle System as simulation density input is to control both the source (Niagara or Cascade) and the target (Ninja) at the same time.

Load a particle system in Niagara or Cascade. In Ninja, pick the same particle system as density input. The particle file should be in the same folder as the preset file. Switch back to Niagara / Cascade using [Shift +F1] or [Alt+Tab].

The dual setup is responsive: changes made in Particle Editor are instantly visible in Ninja (in the same cases the particle system should be saved to refresh the Ninja-connection). Switch back and forth, tweak parameters simultaneously using both systems.

When finished, save both the Ninja preset and particles, and start experimenting with baking.

Status

Tooltips

FluidNinja

NiagaraXraySun

Canvas

Size 1,024 1,024

Offset 0.0 0.0

Scale 1.0

Rotation 0.0

Bake

FrameRange 16.0

Prefix

Rec. every 5 frame

Loopback 0 frames

Rec. mode atlas

masked

Velocity

Offset X 0.0 Amplify 10.0

Offset Y 0.0 Turbulence 1.1

Rotation 0.0 Noise 0.3

Density

Input Weight 0.16 Out Weight 0.72

Shading 0.0 Contrast -0.07

Hue 0.36 Iterations 6.0

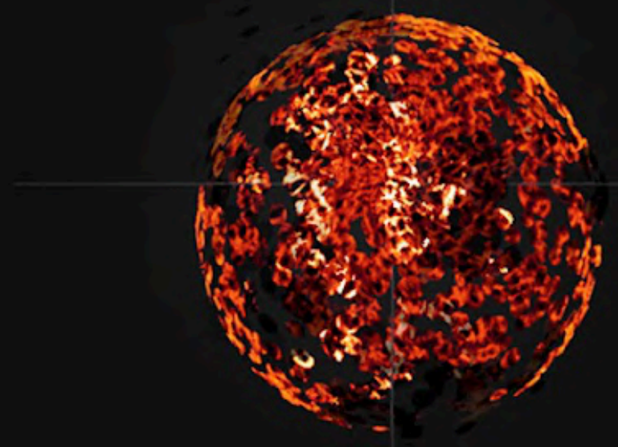
Sharpen -0.25 Shrp.Size 5.0

Input

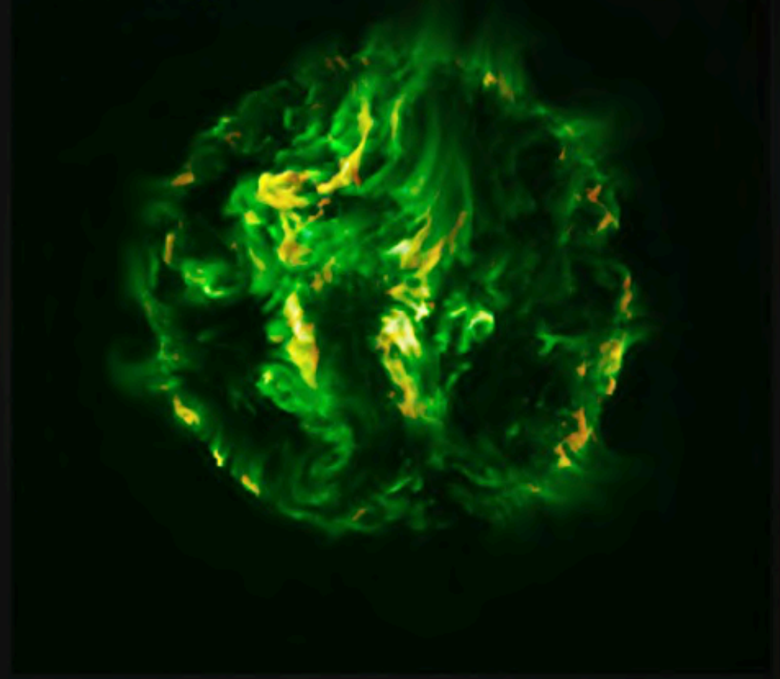
Density from Particles

NS_XraySun

INPUT Density



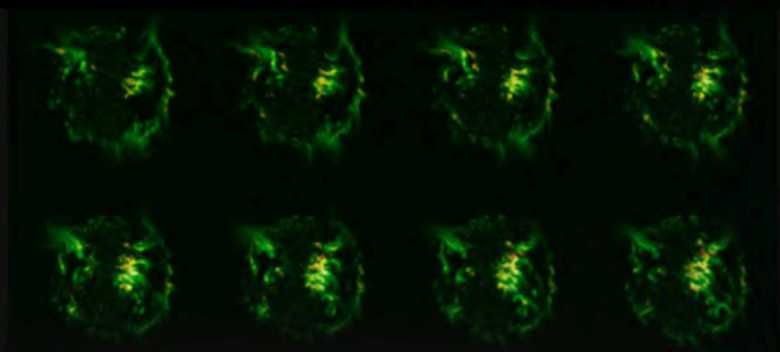
OUTPUT Density



Sequence Player

X

BAKED Sequence



DRIVE FLUIDSIM WITH NIAGARA

USE NIAGARA AS FLUID SIMULATION INPUT IN NINJA CORE

Frame blending

TEXTURE UASSET
Mark baked data for save - if looks good

MATERIALS
Create container material for baked data

PNG
Export data

Input: bitmaps CHAPTER 12

Density from Bitmap
In this roll down menu, you could pick an image (Texture2D) as density template

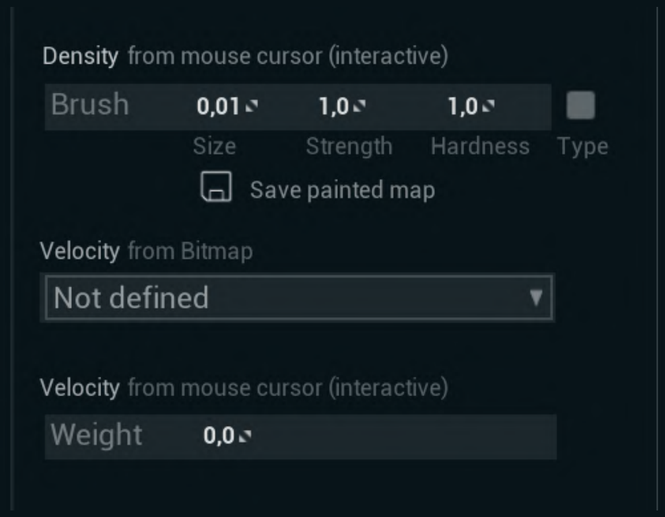
1. Use existing bitmaps as density source (The map must be in the same folder as the preset to be accessible)

2. Interactively painted maps could be instantly saved as bitmaps (see the disk pictogram below) and reloaded here. Do not forget to save the preset once bitmap input is defined

Input: NinjaPaint CHAPTER 13

NinjaPaint is not a separate module but an integrated function of FluidNinja, meant for rapid prototyping: with this realtime painter channeled to the simulation space, (1) you could test various spatial density configurations before you start building a dynamic particle system for the same purpose or (2) you could finalize the simulation setup, save the painted texture and reload it as bitmap input - this way, you can save the complete result of your improvisation as a preset (both the painting and the simulation params related to it). Remember: when using NinjaPaint, the newly created painting should be saved as texture before quitting Ninja in order to be available later.

To enable NinjaPaint set both *particle* and *bitmap density inputs* to "not defined". In this case, the FluidNinja module starts using "Density from mouse cursor". You can adjust the brush parameters at the same place, under the input options. You can save and reload the painted bitmap here. Try adjusting the *weight* of "velocity from mouse cursor". Check the “Freehand” preset group, demonstrating the above config.
Tutorials on NinjaPaint: video link1, video link2



Interactive brush params

Set "Input weight" on the Density Tab to 1, to paint persistent patterns, otherwise the painted spots are fading out / dissolving
Hold [SHIFT] while drawing to ERASE
Use the "Clear Canvas" button on the Canvas Tab to blank the drawing area

Type: line drawing brush VS discrete dots brush

Save Painted Bitmap: save your interactively brush-painted density input as a bitmap, to make it accessible for further use (otherwise the painted data is lost on quitting Ninja). Once saved: erase / blank canvas, and go to the "Bitmap as density input" roll down menu above, and load your freshly painted / saved map. Important: to paint a persistent (non dissipating) image the "Input Weight" param must be “1”. When Re-loading the painted density as bitmap, the input weight must be lowered, to avoid extreme density.

Velocity / weight: by default, you are generating only density data with the interactive brush paint. By increasing the value of this param, you are injecting the brush velocity information to the sim. This feat is hinting a future ninja version (NinjaLive), where users could place interactive simulation containers in game.

Record FGA data in FluidNinja

CHAPTER 14

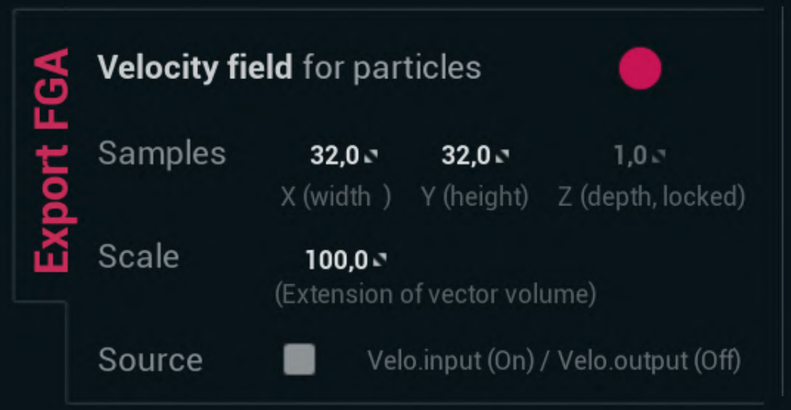
Capture the current simulation frame / actual state of velocity field
as FGA (FieldGridAscii) type data (vector field data), to drive GPU particles.
Depending on FGA-export / plugin settings in NinjaConfig, the data is...

a.: saved in Unreal Editor as DataTable, that could be
converted to FGA in a few steps
b. exported outside UE as FGA and could be dragged back in one step

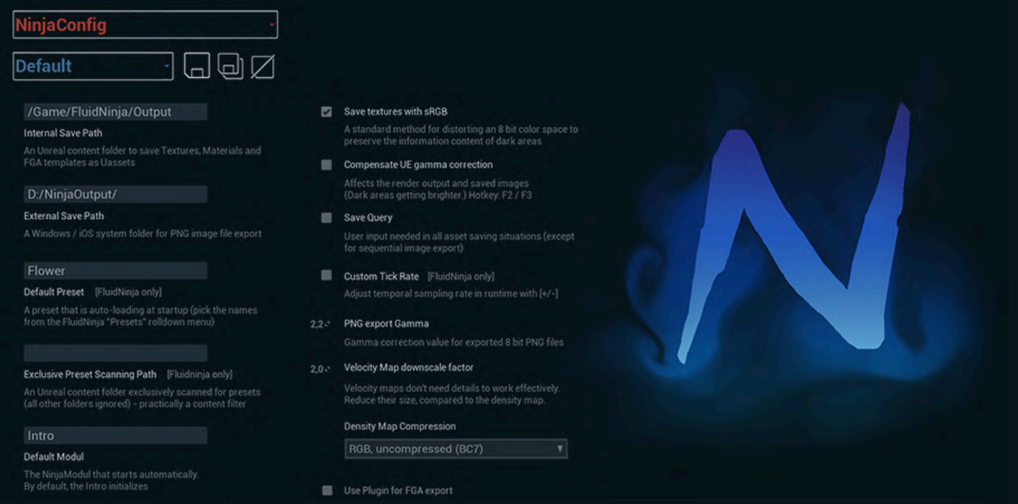
Note: while FluidNinja is capable of capturing velo data, in most cases we recommend NinjaFields - a dedicated module to capture single and multi frame velocity data.
For more info on FGA creation check the “NinjaFields” and “FGA export” chapters.

Using NinjaFields, in details: video link1, video link2

+ on page 22 in this document



Samples X, Y: set the spatial resolution of recorded velo. field, 32-to-128 is ideal range
Samples Z (locked to 1): to record multiple frames as a vector field array, use NinjaFields
Scale: set the UE-world unit scale of the recorded field. This setting could be adjusted later, the scale of the field could be changed anytime
Source: choose between velo input and velo output as the source for vector-field-recording



NinjaConfig module

CHAPTER 15

A module to set defaults, manage asset saving and exporting, gamma correction, sRGB color space usage, image compression and many more. Tutorial: video link1 , video link2

InternalSavePath
An Unreal content folder to save Textures, Materials and FGA templates as Uassets

ExternalSavePath
A Windows / iOS system folder for PNG image file export

DefaultPreset
A preset that is auto-loading at startup (pick the names from the FluidNinja "Presets" roll down menu)

ExclusivePresetScanningPath
An Unreal content folder exclusively scanned for presets - practically a content filter

DefaultModule
The NinjaModul that starts automatically. By default, the Intro module initializes.

NinjaRootFolder
A folder containing all Ninja assets. By default it is /Game/FluidNinja
(same as Content/FluidNinja)

SaveTexturesWith_sRGB
A standard method for distorting an 8 bit color space to preserve the information content of dark areas

CompensateUEGammaCorrection
Affects the render output and saved images (Dark areas getting brighter.) Hotkey: F2 / F3

SaveQuery
User input needed in all asset saving situations (except for sequential image export)

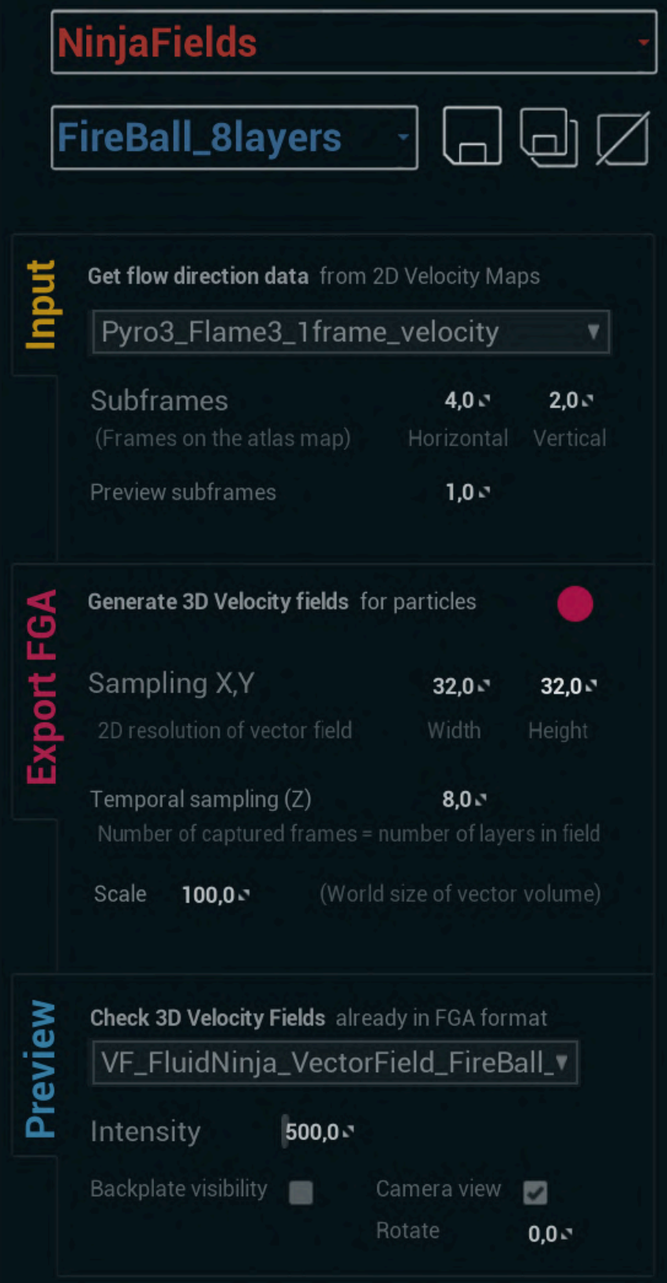
CustomTickRate
Adjust temporal sampling rate in runtime with [+/-]

PNGExportGamma
Gamma correction value for exported 8 bit PNG files

VelocityMapDownscaleFactor
Velocity maps don't need details to work effectively. Reduce their size, compared to the density map.

Use Plugin for FGA export
The conversion of Ninja generated velocity data to UE VectorFields could be simplified by using an arbitrary third party plugin that could export a string variable as plain text (eg. EasyFileSystem or Rama)
Ninja is prepared to work with these plugins. To enable plugin usage:
1. flag this setting, 2. place the plugin blueprint node to:
/Game/FluidNinja/BP_Define_3rdPartyPlugin_for_VelocityFieldExport

More info on FGA export plugin usage: video link
More info on FGA export: see previous page and page 23



NinjaFields module 16

Input

Pick a velocity map for FGA conversion. Could be a single image or multiframe Atlas map

Subframes: provide the number of (sub)frames of the above defined map. (1x1) means it is a single image, not animated

Export FGA

Capture one / more frame(s) of the loaded velocity map as FGA type data (FieldGridAscii) to drive GPU particles. Depending on FGA-export / plugin settings in Config, the data is...

a./ saved in Unreal Editor as DataTable, could be converted to FGA in a few steps
b./ exported outside UE as FGA and could be dragged back in one step

For more info on FGA creation check the dedicated FGA export chapter.

Sampling X,Y: set the Horizontal and Vertical spatial resolution of recorded velocity field
32-to-128 is the ideal range

Temporal sampling: set the Depth (Z) spatial resolution of recorded velocity field.
Depth is interpreted as an array of subframes: the tiles of an animated texture (Atlas) are stacked along the Z axis. 1 is the default value - for anim textures 2-to-16 is the ideal range

Note: the generated vector field will have a depth even if Sampling Z=1. Z samples are meant to introduce field variations along the Z axis, and not to spatially extend the Z axis. The spatial extension on XYZ is uniform (the generated field is cuboid) and could be scaled with the parameter below.

Scale: Set the UE-world unit scale of the recorded field. This setting could be adjusted later (the scale of the field could be changed anytime)

Preview

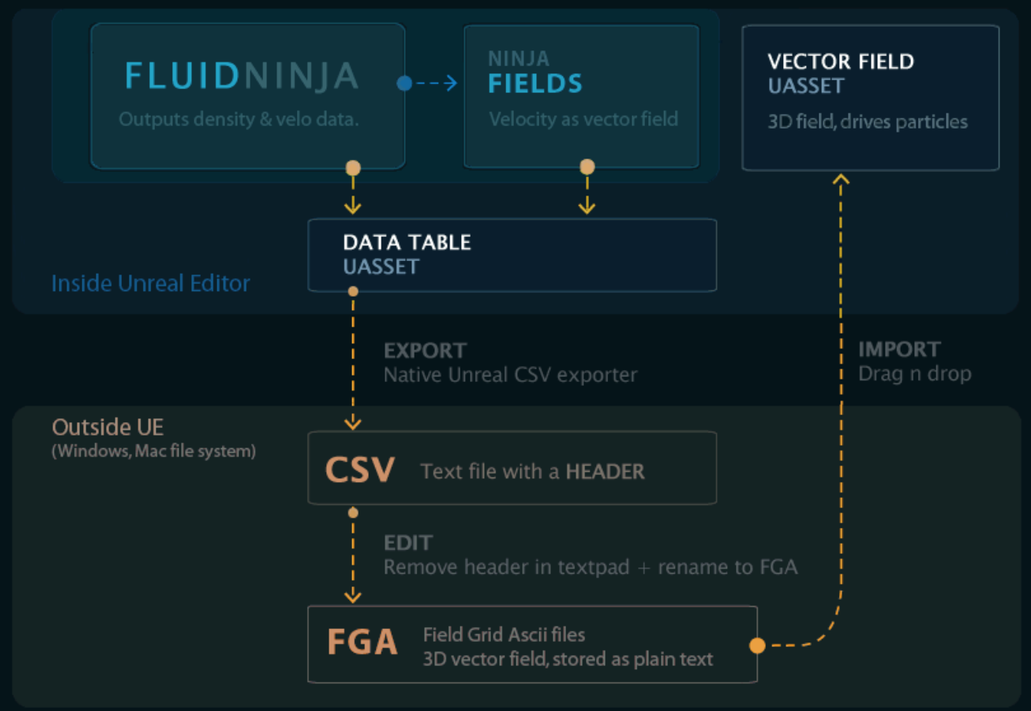
Pick an existing VectorField to preview. Knowhow:

1. a velocity map (texture) goes through the FGA creation process
2. the FGA is pulled back to Unreal and exists as a VectorField Uasset (from this point, it can be used to drive GPU particles)
   (see the Ninja modules data flow chart + details in FGA export chapter)
3. VecField Uassets should be placed on Level in order to preview in NinjaFields. The placement is automated: in the editor, click on the NinjaFields blueprint. On the "Details" panel, there is a button: "Build Vector Field Actors for Preview". Once done, the newly generated VecFields show up in the preview roll down menu.

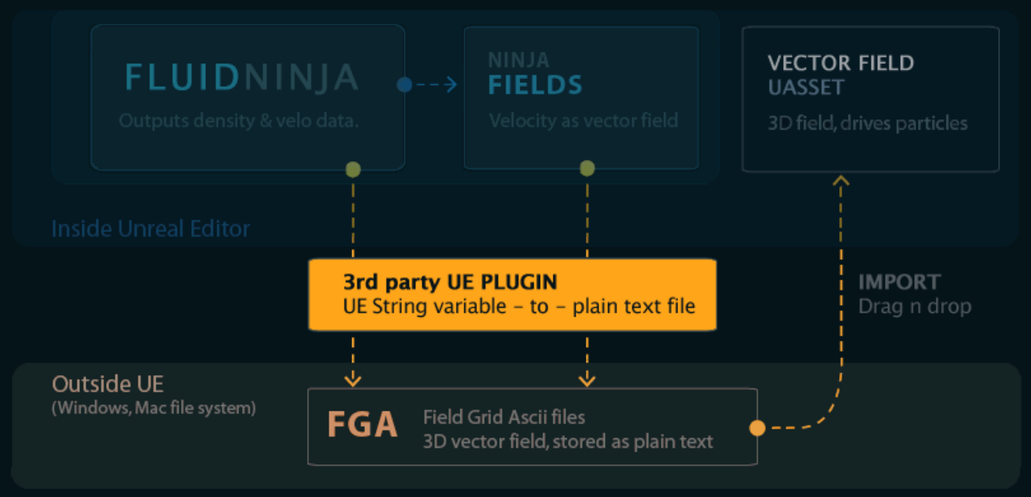
Intensity: scale the field vectors
Backplate visibility: if an Input velocity map is defined, it could be previewed together with the GPU particles. This visualisation helps to understand how fields influence motion.

Camera view: change from Top view to Perspective Camera
Rotate: fly the Camera around the system

A
method



B
method



FGA export

CHAPTER 17

See the data flow chart in the Modules chapter

FGA (Field Grid Ascii) is a generic, plain text format to store an array of 3D vectors. Unreal recognizes FGA files and creates a StaticVectorField object on import. VectorFields drive GPU particles. Ninja could convert simulation velocity data to FGA format, as an initial step to create UE vector fields. There are two methods to export the data to an actual FGA file.

Method A (video link1)

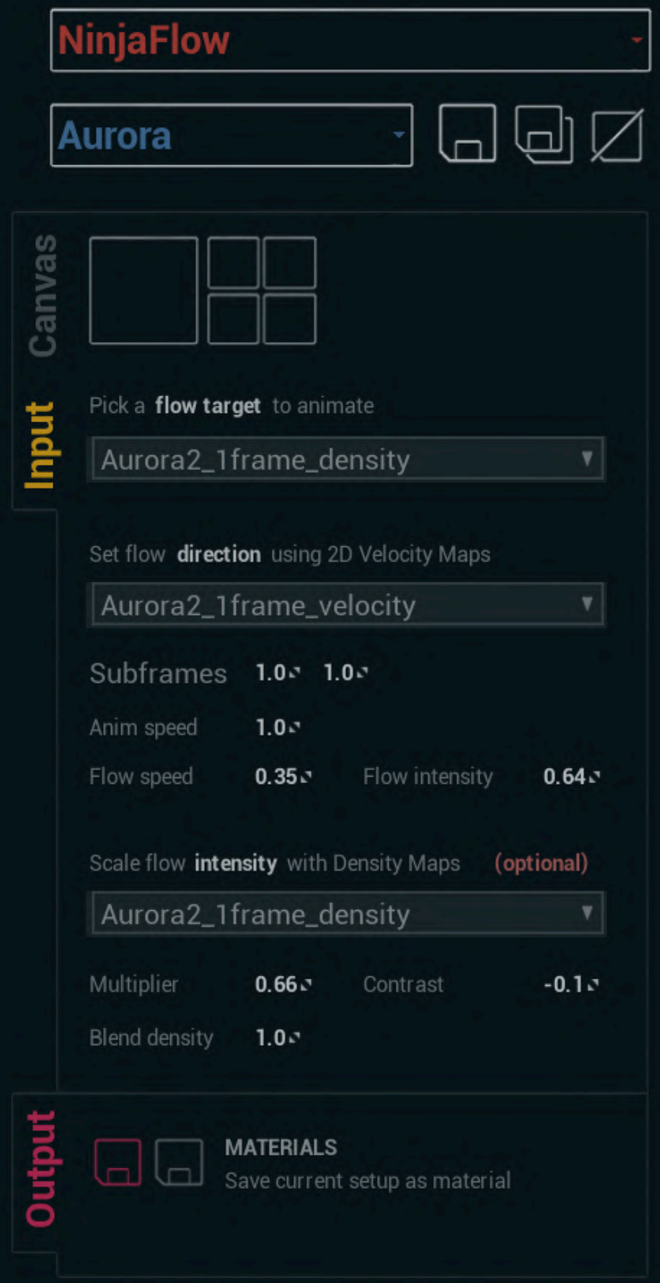
1. save the fluidsim velocity to a DataTable type Uasset by using the REC button in FluidNinja or NinjaFields
2. export the DataTable to external file (quit Ninja, in UE Content Browser right click on the saved DataTable asset, and choose "Export as CSV")
3. remove the CSV file header (open the exported CSV file with textpad and remove the initial 24 characters: [---,DataColumnDataRow,"]
4. rename the file from CSV to FGA + make sure that the filename is different than the original DataTable name, otherwise UE will try to update the original Uasset
5. import / drag the renamed file to UE Content Browser

Method B (video link2)

An arbitrary, 3rd party "UE string variable to text file" export plugin is needed.
Eg.: "Easy File System" or "Rama"

1. install / enable the plugin (Menu/ Edit / Plugins)
2. Ninja is prepared to work with these plugins. To enable plugin usage, flag the "Use plugin for FGA export" option in *NinjaConfig* + place the plugin`s blueprint node to: BP_Define_3rdPartyPlugin_for_VelocityFieldExport

From now on (when pressing the record FGA button) an external FGA file is generated, that could be instantly imported / dropped to Unreal (no need for editing)



NinjaFlow module 18

See FlowPlay chapter for description on the possible usage of flowmaps!
Using NinjaFlow, in details: video link

Input

Flow target: pick an arbitrary texture that you would like to animate via FlowMaps.

Set flow direction: pick a single or multi frame velocity map to animate the target texture. Already baked velo frame(s) are used to distort the target image in realtime, via the FlowPlayer material.

Subframes: define the number of horizontal / vertical subframes. (1x1) means it is a single image, not animated
Anim speed: the play speed for animated velocity/density maps

Flow speed / intensity: the pixels of the flow target texture are pushed with this speed and intensity along a velocity defined trajectory

Scale flow intensity: pick a single or multi frame density map for modulating the target texture animation. Density maps are optional, FlowPlayer could work without them.

Density maps could be used two ways:

1. Mask velocity (dark areas will have lower / zero velocity)
2. Mask opacity: this is NOT previewed in the NinjaFlow module

Multiplier: multiply the velocity driven distortion of the target map via density

Contrast: apply a power function on the density map (a bit like a contrast filter). Sub 1 values brighten / flatten the map, values above 1 are increasing contrast between dark and light areas

Blend density: Crossfade the density map and the flow target. 0.5 means 50% blending

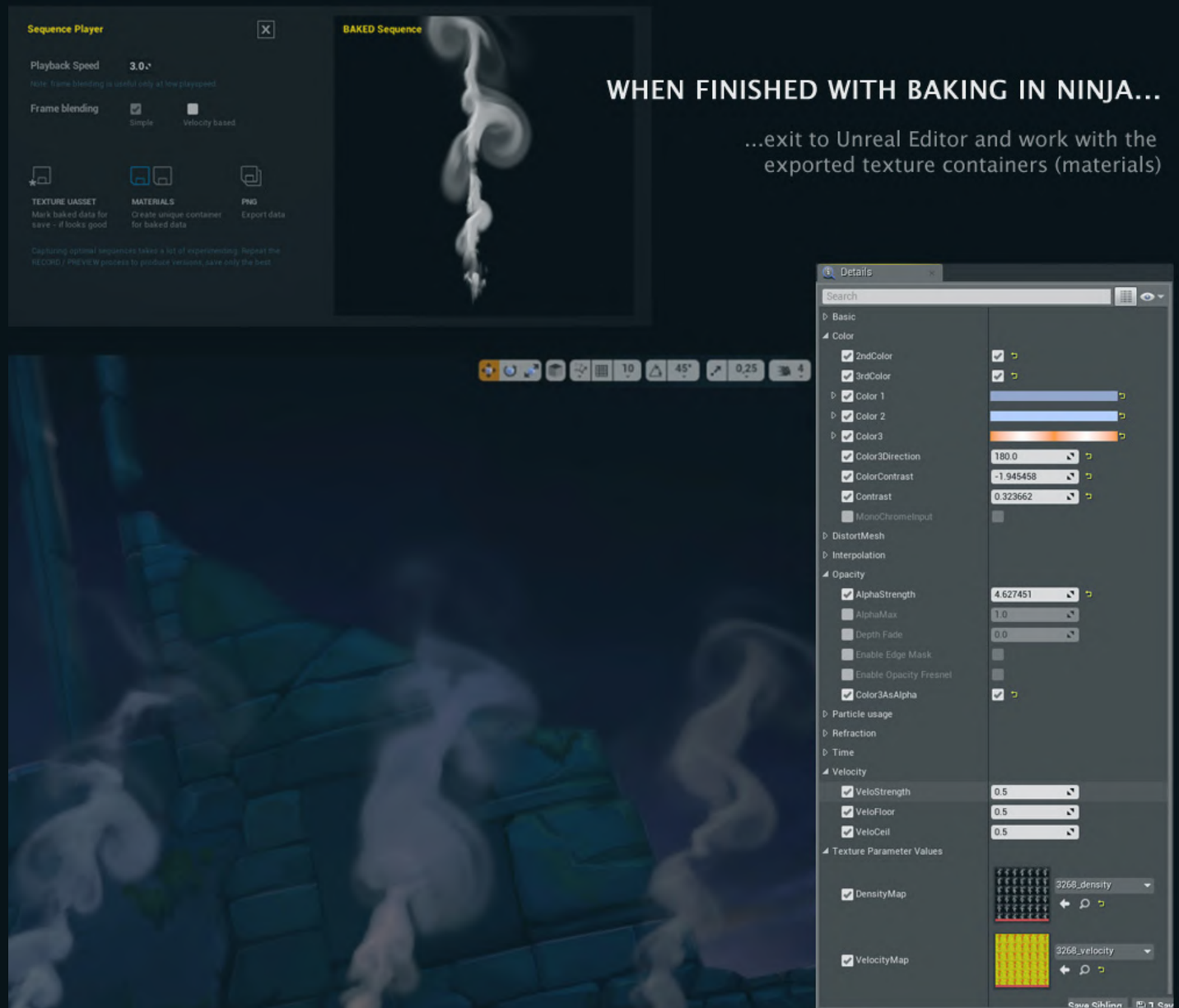
Note: the NinjaFlow GUI is meant to link maps together and save them as a material - as the first step of VFX development. The preview is very crude here (no transparency, no velocity based frame blending, no coloring). The major part of visual dev. is based on the exported FlowPlayer material - after quitting Ninja. When finished with the basic setup of a FlowPlay material, save it (the disk pictograms below), quit Ninja and continue the visual dev. on the FlowPlayer material interface in the editor.

Output

Save an advanced material (FlowPlayer) with the above settings.
This will be the base for further visual dev, after quitting Ninja

Save a basic material (FlowPlayer) with the above settings.
Gives approximately the same look as you can see here
in the viewports (no opacity mask, no velocity interpolation).
For preview purposes only

Tweaking baked data CHAPTER 19



Working on the Ninja GUI is only the first part of VFX development. When finished, we have produced:

1. simulations, baked as textures
2. advanced materials, referencing the baked textures
3. vector field data to drive particles

Quit Ninja. Apply the container materials to static meshes, particles and blueprint components. Drive Niagara Systems with baked data!

Tweak material params to set the opacity and color of baked data. Set the play speed and velocity params. Enable refraction (think heat distortion) and geometric distortion in the materials.

Try to enrich the baked data by adding in flowmaps. Match colors.

Use more baked components together:
eg. flame + smoke + GPU particles (sparks)

See the content examples level for examples on material usage and combining various baked elements.

Check this video on the features of Container Materials: video link
+ a discussing the usage of baked data on Cascade particles: video link

Additional Content CHAPTER 20

Pack1: Exploring Raymarching and Parallax Mapping technology

- *Included* in Ninja v1.1, *(released on the 8th April 2020)*
- Also available as free, downloadable package *(see below)*

1. Unreal Project documentation:
   Adding Depth and Realism to 2D Fluids in Unreal
   PDF LINK, 80.lv article LINK

2. Two tutorial videos, 13 minutes, at Youtube
   https://youtu.be/-gDVyrPyvEs
   https://youtu.be/TUk4sytRpfA

3. Downloadable, Zipped Unreal Project (UE 4.20 and above), *45 Mbytes*
   Mirror1, LINK
   Mirror2, LINK

The project is a collection of blueprints, materials and baked data, arranged on two demo levels - delivering technology to make 2D flipbooks “look 3D” by adding depth and self shadows to simple fluid simulations.

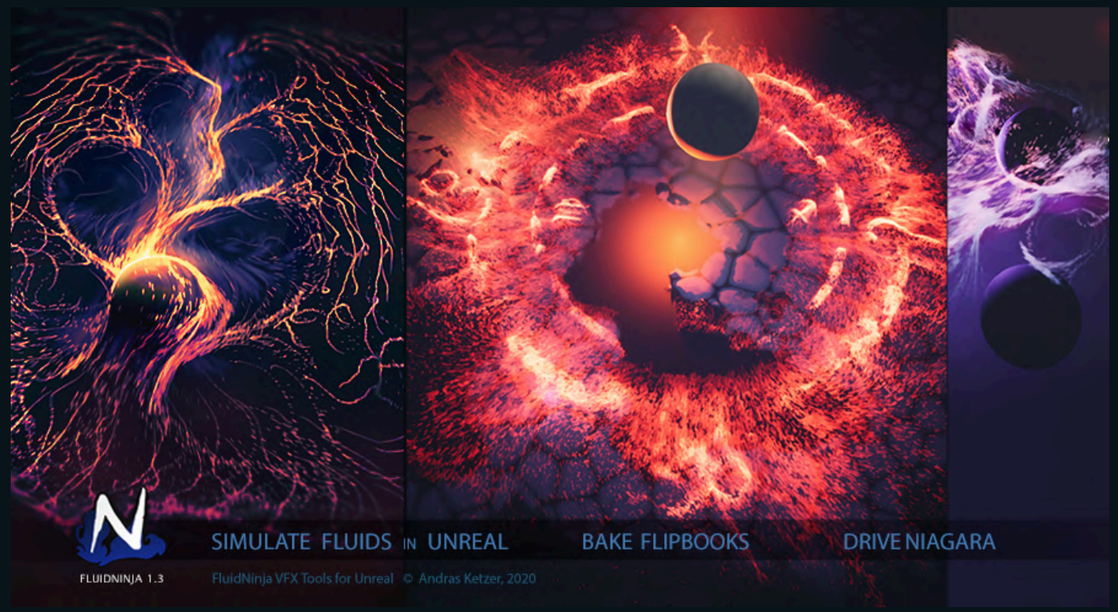
If you are considering using Ninja: this project is a good intro with example content, baked data and materials - have a look! It is free to use and could be utilized without Ninja.

For standalone Flipbook player, see: NinjaPLAY

Niagara CHAPTER 21

Driving Niagara with baked fluidsim: Ninja 1.4 was released with custom Niagara modules and Base Emitters - used to build Niagara Systems that play flipbooks autonomously / do not rely on blueprints.

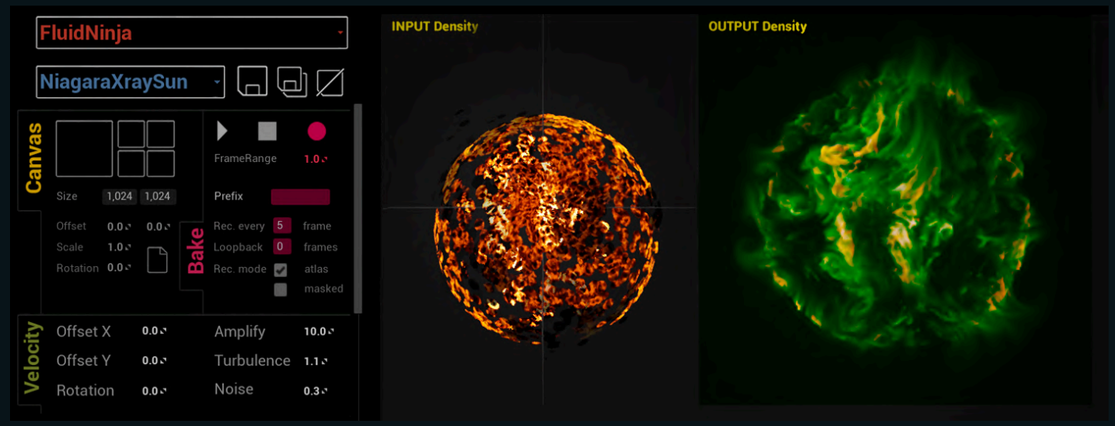
Dedicated manual: LINK



The 30+ example systems included in Ninja v1.4 are combining generative and pre-rendered VFX developing approaches: besides imposing constraints on particles, the baked fluid data is used in an additive manner.

As a result of generative approach, a given system could be pushed towards stylized or realistic look via high level tweaking, no need to change the core fluid data.

Ninja could read Niagara Systems (see chapter 11) to generate density input for simulations - and drive Niagara GPU particle systems with baked fluid data: there is true two way data flow between Ninja and Niagara.



The following block is a set of legacy instructions to manually enable Niagara reading support in lower Ninja versions, running under UE 4.20 - 4.24.

FluidNinja simulations could be driven by particles. By default, Cascade Particle Systems have been used as input. An optional update has been released that enables Ninja to read Niagara Systems. The update should be performed manually, by downloading a zipped package and overwriting three files of the original Ninja project.

1. Start UE. Load Ninja. Enable the official Niagara plugin (Top menu / Edit / Plugins)
2. Save Ninja. Quit UE
3. Copy the packaged files to the *Content* folder of the Ninja project.
   The update could be downloaded here, 2 Mbytes: link

Volumetrics CHAPTER 22

VOLUMETRICS V2

Unreal Engine 5.4 came with a new volume type: Heterogeneous Volume.
Accordingly, NinjaTools has been updated with a new generic volume function,
Handling **ALL** native UE volume types: *fog*, *clouds* - and *heterogeneous volumes*.

The update comes with…

- wrapper materials for the new volume function
- minimal setups, showcasing volume function properties
- flipbook and static-texture driven setups

Setups are demonstrating usage in the 1 meter to 100 kilometers size range,
combining various volume types and exploring multilayer volumes.

COMPLETE VOLUMETRICS V2 DOCUMENTATION: LINK

NEW CONTENT: */Content/FluidNinja/Usecases/Volumetrics_v2*
NEW LEVELS: */Content/FluidNinja/Levels*

The update also employs a new technique to align volumes with surfaces.
We are sampling landscape height using RVT volumes and forwarding the height data to the volume material.

RELATED TUTORIAL, LANDSCAPE ALIGNED VOLUMES: LINK

VOLUMETRICS V1

Example content levels: *VolumeSmoke, VolumeFog, VolumeClouds, Tiling*
Tutorial Video: YouTube

NinjaTools 1.5 - 1.6 could drive three volumetric systems with flipbook baked 2D fluid sim: users could set up sim containers on level or place a single VolumeCloud Actor for the sky. Baked density is combined with 2D & 3D noise. Output volume is shaded by scene lighting, and looks good from all camera angles. Comparing the three systems:

	FOG	SMOKE	CLOUD
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	-	-
Frustum aligned grid*	True	-	-

Note1: UE native volume fog is implemented as **froxels* - voxel array stretched by the cone-shaped camera frustum. This means the relative (per square unit) resolution of the grid is getting lower by distance. For this reason, NinjaLive VolumeFog Base Material contains a “fade by distance” function - highly advised to use it.

Note 2: the spatial and temporal resolution of fog could be adjusted by console commands, listed on level 23. r.volumetricfog.gridpixelsize is the most important - by default = 8 ---> set to 6 by the *Level Blueprint* on the VolumeFog demo level!

22.1 SETUP COMPONENTS / SETTING UP A VOLUMETRIC SYSTEM

A. Flipbooks: generate and bake flipbooks using the NinjaTools level - or use one of the existing flipbooks at */FluidNinja/Usecases/AnimTexturesAndMaterials/*

B. Materials: volumetric systems need a *flipbook player material*, providing input - AND a *volumetric shader material*, visualizing the data (output). In the case of Fog and Clouds, the player and the volume shader are *integrated* to a single material - while VolumeSmoke uses separate input/output material pairs.
Base Materials are located at: */FluidNinja/Usecases/Volumetrics/BaseMaterials/*
In practice, we are using *Instances* of the Base Materials:
*/FluidNinja/Usecases/Volumetrics/Instances_*.**
Duplicate one of the existing player mat.instances and feed it with the new flipbooks.
Optionally, you could generate a Player Material Instance together with the flipbook, and *re-parent* the instance to one of the above base materials.

C. A level placed Actor is used to call materials, define volume bounds and position

22.2 DEFINING VOLUME BOUNDS & POSITION

A. Volumetric Fog: a simple *StaticMesh Box* is placed on level, equipped with Volumetric Domain material, reading *flipbooks*. See *"VoluCube'' meshes* listed in the *WorldOutliner* and placed *on Level*.

B. VolumeClouds: bounds are defined within the cloud material (no mesh needed), using UVW coordinate offset and scaling. Cloud material is defined by a single, special, level placed actor: Volume Cloud Actor

C. Volume Smoke: bounds are defined by a level-placed blueprint: *VolumeSmoke Container*. A debug box visualises volume extension. To resize the volume, simply scale the blueprint actor - use uniform scale. Volume location is anchored to the blueprint actor. Define in/out materials on the Actor Details panel.

22.3 FLOW / SPEED

Clouds and Smoke systems could use a user defined detail map (usually noise) imposed on flipbook data. Flipbook stored *velocity* is used as a flowmap to advect the detail map. VolumeClouds uses a 3D volumetric noise - while VolumeSmoke could employ 2D noise in the input material AND 3D noise in the output material - with different performance impact. See example levels for more details / description.

The below listed settings are located in the VolumeCloud/Smoke Material Instances

A. Adjust Noise Flow Strength, Speed and Weight in the NOISE param group to influence the drift of *small* structural details

B. Adjust adjust Playback Speed in the FLIPBOOK or BASIC param group to influence the tempo of *large* structural changes

C. Turn on and adjust velocity based frame blending in the FLIPBOOK param group to support super-slow-down

Ninja could use baked simulation velocity data to create smooth blending between density flipbook frames - when playback is slowed down. To enable this feat, set Velocity Interpolation = TRUE in the PlayerMaterial (read more: chapter 4).

Velocity-based frame blending is "pushing pixels'' towards velocity direction. The amount of optimal "pushing" varies, depending on the play speed. Following the baking process, we could manually adjust velocity settings (Strength, Floor, Ceil) in the player material, according to the playback speed.

22.4 TILING SIM SPACE: FULL SKY / GROUND COVERAGE

Tiling enables us to pre-calculate (bake) only a small part of the sky (or ground-fog) - and use this part repeatedly - the sim space is wrapped - so, we could cover the whole area (sky or ground) using a single pattern.

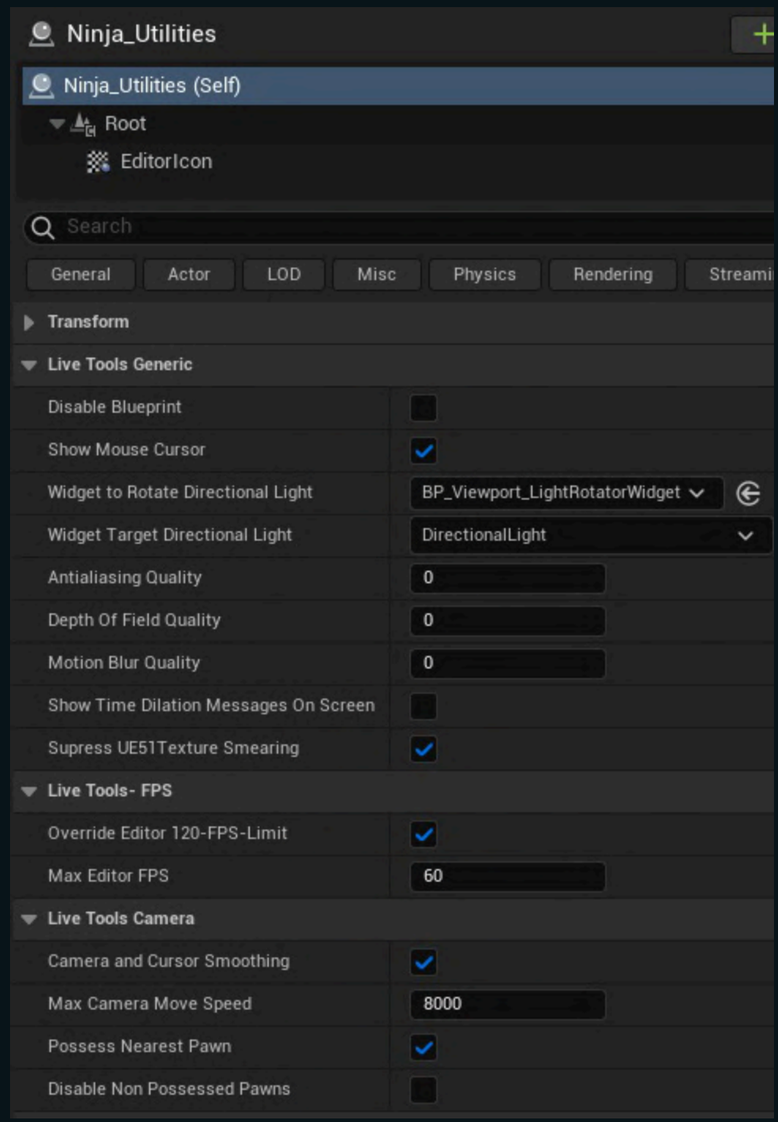
Tiling VS Flipbooks: Ninja stores flipbook frames in a compact, more-or-less square-like grid structure (eg. 4x4 grid to store 16 frames). The default texture smoothing of video hardware (bi-linear interpolation) is blending between adjacent pixels - too bad it does the same at frame edges: border pixel values of a given frame are corrupted by the values of the pixels from the surrounding frames.

This is not a problem, in case we have useful information only mid frame - eg. a candle flame sequence. This is a problem, in case we would like to crate full-frame tiling patterns - typically needed to cover large areas (eg. a cloudscape) with repeating flow patterns. Ninja uses a workaround to avoid "corruption" caused by the video hardware interpolation - but, it is not a perfect solution: frame edges are somewhat glitchy.

Simple solution: offset the tiled pattern until the artefacts are not visible :)
Complex solution: bake the sequence to a long, linear stream of images, so frames will not have vertical neighbours (while horizontal neighbours are temporally close to the current frame). The result is a flipbook with one row - the number of columns = the number of frames. Ninja looks up grid-creation settings from a data table. By manually adjusting this table, you could force ninja to create such long flipbooks:
/Content/FluidNinja/Core/AtlasBuilder/DT_AtlasMatrices

Eg.: you'd like to bake a loop to 36 frames. Ninja, by default assigns a 6x6 matrix to this frame number. You should set it to 36 x 1 in the atlas matrices datatable before starting the baking process. As a result, Ninja will generate a single row flipbook - somewhat resistant to tiling artefacts.

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



FluidNinja_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. For WASD Pawn control in merged projects, see this PDF: LINK

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