Grenoble INP – ENSIMAG
École Nationale Supérieure d'Informatique et de Mathématiques Appliquées

Engineer Assistant Internship Report

Carried out at Eviden Atos

Visual Effects for Synthetic Data Generation

HASSAN Hélène
2nd year – MMIS option

22 May 2023 – 31 August 2023 (15 weeks)

Eviden Atos
3 rue de Provence
38130 Echirolles, France

Internship supervisor
DEVEZE Louis
School tutor
DUMAS Julie

Contents

I Introduction

Nowadays, public places such as train stations or airports are likely to endure a large variety of incidents in the midst of the daily visitors going through them, ranging from simply losing a luggage or a bag to experiencing thefts, fires or even floods. These events, often resulting in frustration or despair on behalf of the victims or public security workers have always been a hard challenge to overcome. As a result, public institutions have increased their need for tools to detect and prevent these situations.

Eviden, the company in which I have performed my internship, is tackling this particular challenge amongst many others by developing a custom-made solution that uses security cameras coupled with AI models trained specifically to identify such incidents. Nonetheless, finding sufficiently large data-sets of good quality to train or test AI models can often be quite difficult. In order to work out this problem, the Synset team -that belongs to the Research and Development pole of Eviden in Echirolles-, provides an application based on the Nvidia's Omniverse ecosystem to generate large and realistic computer generated data-sets.

During my internship, I joined the Synset team in order to add visual effects to the generated images inside the synthetic data-sets. In the course of this 15 weeks journey, I mostly focused on fire which is mandatory within the scope of blaze detection and later on rain which is required to add realism and quality to the data-sets.

II Background

A Eviden, an Atos business

Founded in 1997 through the union of two French IT (Information Technology) companies, Atos is a worldwide corporation specialized in IT services, meaning that it provides custom-made end-to-end solutions for industries in 69 countries. The company currently employs around 110 000 people in the world and stands as a global leader in secure and decarbonized information technologies.

Atos activites are divided into 3 categories. The first one is **Big Data and Security (BDS)** which develops high-growth projects related to artificial intelligence, cybersecurity, supercomputing and quantum computing. The **Digital** section puts together projects belonging to decarbonization and digital transformation. The final part of Atos is **Tech Foundation**, which is focused on designing, building and managing information systems. This field belongs to Atos' earliest business.

Recently, Atos split into two companies: **Eviden**, which holds the BDS and Digital sectors and **new Atos** that keeps running the Tech Foundation activity.

Figure 1: *A picture of the Iseran campus. // source: Google images*

My internship took effect at the Iseran Campus localized in Echirolles, France. Inaugurated in october 2022, this campus is the second largest site in France and is known as a leading research and development (R&D) center in Europe. It brings together three main fields of activity:

→ **WorldGrid**, which activities address energy operators through flexible, durable and ecological solutions.

→ **Technical Solutions** responsible for providing advanced technological solutions such as integrated systems, consulting services or software developments which supports digital transformation of the companies in the Auvergne Rhone-Alpes region.

→ **BDS (Big Data and Security)**, which holds teams specialized in HPC (High Performance Computing), Artificial Intelligence, Cloud and Cybersecurity. This division also provides solutions ranging from integrated systems to software development or maintenance. The R&D part of the campus takes place inside the BDS division.

B The CVLab (Computer Vision Laboratory) and Synset

I carried out my internship in the CVLab, an AI R&D team belonging to the BDS division which is currently working on incorporating AI into surveillance solutions as a way to improve their efficiency. The CVLab is divided into 3 sub-teams working very closely to each other:

→ The **Model** team is responsible for creating and training new AI models based on Deep Learning algorithms that will be integrated in concrete solutions afterwards by the **Innovation** team.

One issue with the models based on deep learning is that their accuracy is highly dependant on the quality and quantity of the information provided by the data-sets. However, creating massive data-sets with real information can be quite challenging given the nature of the data itself and its use that may be prohibited by the General Data Protection Regulation (GDPR).

In order to cope with this issue, the **Synset** team offers an application which aims at generating large and realistic computer generated data-sets (also called *synthetic data-sets*). The team is also working with European partners in order to research on the reliability of AI models. During my internship I joined the Synset team to help them incorporate visual effects into their application, which is based on the Nvidia Omniverse's suite of tools.

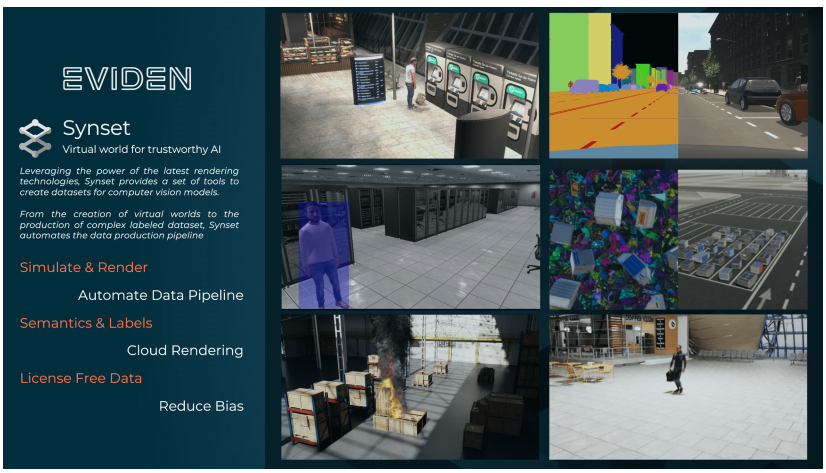


Figure 2: *Overview of Synset activites.* ***a)*** *Top-left and bottom-right images: Avatar and 3d scene generation.* ***b)*** *Top-right and middle-right images : Post-processing AI tools to obtain the segmentation of an image. Third image (middle-left): Human detection.* ***c)*** *Bottom-left image: visual effects simulation. // source: Synset*

C Omniverse and USD

Omniverse is a platform created by NVIDIA in 2021 that contains a set of applications which aims at enabling both virtual and real-time collaboration. The team was previously developing their application using Unity for the visual effects and 3D assets but in early 2023 the engineers of the team decided to migrate to Omniverse in order to benefit from the many advantages of this ecosystem. For instance, Omniverse can display 3D scenes using multi-GPU rendering which enables to represent vast scenes in a minimal time. The platform is also equipped with many cutting-edge features made by Nvidia, ranging from advanced realistic physical simulations to advanced AI tools.

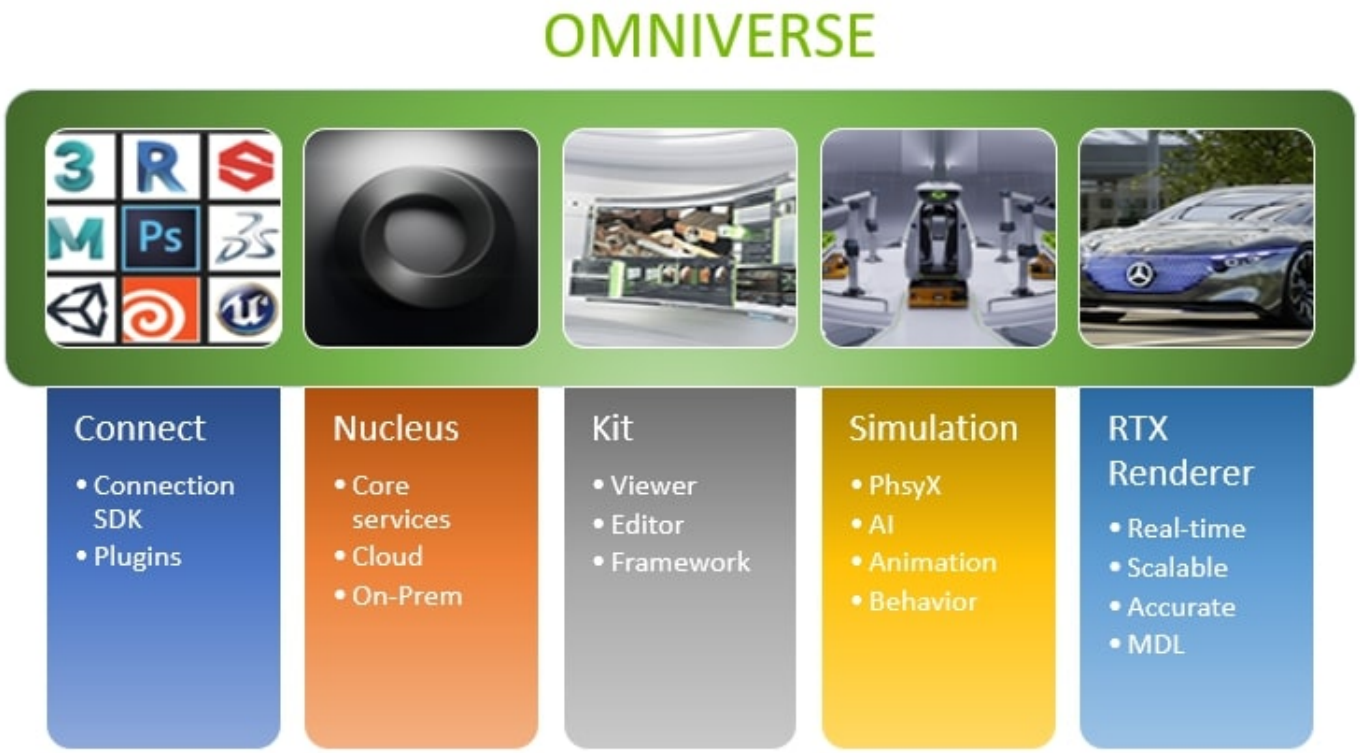


Figure 3: *A description of the 5 main keyparts of Omniverse. // source: NVIDIA Omniverse website*

→ **Omniverse Connect** contains a set of applications that allow users to interconnect their work on Omniverse with that of applications such as Rhino, Maya, Unreal Engine, Revit, Blender and many more. This feature was used by the apprentice in our team who was working on modeling 3D human avatars on Blender and needed to import them in Omniverse.

→ **Nucleus** is basically a database and a space of collaboration which allows users to be connected together live with multiple applications all at once. We used this feature in our team to store our 3D assets, presets and scenes which allowed us to work on it at the same time.

→ **Kit** is the SDK (Software Development Kit) for building Omniverse Applications which is made available for developers to build their own application, microservice, extension or plugin. Omniverse *Code* or Omniverse *USD Composer* are example of Omniverse Kit applications. Kit also comes with a User Interface (UI) engine that helps developers build their own application designs.

An Omniverse Kit Application uses *extensions* which are basically the building blocks of the application. The code inside each extension usually uses Kit and sometimes other extensions to add a specific feature to the global application. For instance, an extension could be as simple as

a window that allows users to write notes within (a notepad) or as complex as a fluid dynamics renderer, or even a UI engine. In practice, extensions do not operate on their own, they instead need the Omniverse Kit Application packaging them together.

One particular extension I extensively used during my internship is *Omnigraph* which provides the ability to create actions by connecting *Nodes* together. For instance, an Omnigraph can be as simple as adding two floats in a node and redirecting the output to a node that would print the result inside a console. It can also be used to build more complex actions, such as skeleton animations, physics simulations and so on. It is similar to the concept of Blueprints in Unreal Engine, for example.

→ **Simulation** represents the extensions of Omniverse that produce highly advanced physics behaviors and animation. Another purpose of Omniverse is to make easily accessible AI tools for the generated 3D scenes, for example tools to label the different objects in the scene or tools that uses AI to generate an entire scene from a text description.

→ **RTX Renderer** represents all the advanced methods used to render objects on the screen (multi-GPU rendering, advanced shadings...).

Finally, one of the particularities of Omniverse is that each application in the workflow highly relies on the use of USD format, which is at the core of Pixar's 3D graphics pipeline.

What is the USD format?

Pipelines capable of producing computer graphics films and games typically generate, store or transmit large quantities of 3D data which we call "scene descriptions". However, each of the applications needed in these pipelines (for example Blender for modeling, Houdini/Arnold for shading...) usually has their own way of storing the data which is tailored to its specific needs and workflow. The way used to represent the data with all the software that enables users to fetch/write into a file is called a *format*. The most common formats encountered nowadays are FBX (Autodesk format), OBJ, 3DS (Autodesk format), BLEND (Blender format)... The main issue with these formats is that they are generally neither readable nor editable by any other application in the pipeline, which represents an important waste of time and data for the users working with them.

The idea behind the USD format is to provide the first open source format that addresses the need to have a rich, common language for describing 3D scenes. In addition, USD comes with a C++/Python API which allows to create USD contents.

USD organizes 3D scenes through the use of hierarchical generic objects called **Prims** (short for "primitives). Prims can be nested and referenced using path much like a file system and contains Attributes and Relationships, which are collectively known as Properties. An attribute possesses a type (string, int, float, array of floats, etc) and a name. Relationships, on the other hand, are path(s) to other prims in a scene. A prim can be of any type, for example: a mesh, a set of points, an Xform, a material or a scope which is simply a folder in which we store primitives to organize our stage.

Within Omniverse context, 3D scenes have been designed to be described through USD terms and concepts. In practice, this means that every scene obtained from an Omniverse application corresponds to a unique USD file that is stored in cache during authoring and that is saved when the scene is also saved.

In order to introduce the project I have done within Synset, some basic notions on USD are required. Thus I linked the Pixar introduction to USD in the annexes. Definitions of the technical

terms or concepts used in this report are also available in the glossary.

III Project

A Overview

A - a The Synset App

The application developed by Synset is an Omniverse Kit Application coded in Python that aims to generate procedural 3D scenes which are already annotated for the AI models that will be trained on them. This application uses several custom-made and built-in extensions.

Extension name	Description
Common	Common tools (UI tools, widgets, callbacks, custom attributes...) used by Synset extensions.
Smart Object	Add additional data and behaviors to USD primitives (see definition in the glossary).
Scenery	Compose and setup simulation stages, either manually or automatically.
Particle System	Simulates and render particle systems. Initially made for the rain effect but can be modified to implement other particle effects.

Table 1: *Table describing some of the custom-made extensions used within Synset. The last one has been added by me in order to display the rain.*

During my internship, I mainly worked on two of the existing extensions: the **Smart Object** extension and the **Common** extension.

→ The Smart Object extension aims at binding additional or simplified data and specific behaviors to USD primitives in the scene in order to make them behave as in real life. For example, a mesh with a Human form will be bound to a Smart Object of type 'Human' and will therefore acquire additional attributes such as the age, the ethnicity or physical attributes (height, weight...) etc. All these attributes, if authored, will have different effects on the object itself such as changing the size of the Human's mesh or the skin color.

The idea behind this extension is to play on the attributes of each Smart Object to bring high-quality content with a wide range of variability and randomness.

→ The Common extension aims at creating UI tools to display properly the content of other extensions. Basically, the extension is in charge of generating re-sizable windows with the appropriate widgets widgets and callbacks corresponding to each type of attributes (float, int, boolean, arrays...). For example, the extension will set for an attribute of type vec3f (vector of 3 floats) a specific widget that displays each component of the vector at the screen and will bind it to a specific callback that will be called each time one of the components is changed.

A - b Internship goals

The objective of my internship was to search for ways to integrate visual effects inside the Synset application, the most important one being the fire effect which will be used to train AI models to detect fire for public security purposes. The need for rain comes from the fact that it adds more realism to a scene. Concretely, this translates into creating two **types** of Smart Objects (Fire and Rain) that will contain all the necessary attributes to represent each effect and

its behavior regarding other objects in the scene.

In practice, I had to break the implementation of each effect into three parts:

1) What are the properties of my visual effect in terms of appearance and interactions with its environment?

→ This part is mainly focused on **defining precisely the properties** of the visual effect we would like to see in a 3D scene and how do we want it to behave with other objects within Synset's scope.

This definition comes when answering to several questions, for instance: *What properties come naturally to my mind when thinking about this effect? Does it change over time? What properties should I add to define its behavior with other objects in the scene?*

For example, if we wanted to implement snow in the Synset app, we would have first detailed the behavior of the snow in itself by defining attributes to control the amount of snow, the color, speed and direction of propagation of the snowflakes and so on. After that, we could have defined properties to represent its behavior when touching certain type of surfaces, for example the melting speed or the density of the snow.

2) Which technologies and/or methods available in Omniverse should I need to use in order to implement the previously defined properties? Do I have restrictions on how to use them?

→ This part stands out from the others as it is much less related to Synset and is more about computer graphics in general. It highly focuses on research and is the most demanding part as it requires an advanced knowledge of USD, Omniverse and 3D rendering in general. During this part, I basically try to answer the following question: *"How do I render my visual effect in Omniverse in a way that all previously defined properties are met?"*.

It is also important to note that the complexity of this part highly depends on both the type of effect we want to render and the properties we defined for that particular effect. Indeed, some effects are more easy to implement with the Omniverse workflow than others. Also, the more complex properties we add to an effect the more we will converge towards a custom-made solution and won't be able to use built-in extensions. The performance costs of each effect is also an important factor that needs to be considered in this part.

3) Which properties are essential regarding my effect?

→ Finally, this part is basically about keeping the essential properties of the effect in order to put them into the corresponding Smart Object type which is, as a reminder, just a way to simplify the description of an object and control its behavior.

To do that, we usually ask to ourselves the question: " *What properties would I like to have an influence on if I wanted to represent all versions of this effect?.*

To pick up the example of the snow again, we know that the amount of snow or its speed are example of properties we would like to keep as they help represent several cases of falling snow such as huge storms, calm snow, slightly windy snow and so on. Properties that don't need to be made accessible for the user are usually related to internal computations, for example the time step we would use to update the state of the snowflakes.

B Issues

Fluid simulation represents an important part of computer graphics and has always been conducive to many technological improvements over the last decades. Given the amount of realism one can require, this type of visual effect can become extremely difficult to process and render.

In the case of Synset, we want to generate data as close to the reality as we can for the future AI models that will be tested/trained on it.

Such realistic simulations usually rely on heavy mathematical computations and cutting edge GPU development, which often takes long time to develop, debug and optimize. By the time I started my internship, I had experiences with realistic fluid simulation but I was always very close to the GPU as I intensively used shaders in order to reach my goals. In the context of Omniverse, there was no sufficiently evolved way to build advanced effects as the rendering pipeline was not made directly accessible for users to add their own shaders. Finally, as rendering each effect was just a part of the work I had to produce, I had to find the more efficient way to obtain the amount of realism we wanted so that I could have time to implement my solution inside the Synset App.

In addition, the creation of Omniverse dates back to late 2021 which is quite recent and brings about many challenges regarding the topic of my internship.

→ First of all, as the development of Omniverse was still in progress by the time I started my internship, there were yet many bugs/errors/crashs inside the applications of the platform (**Omniverse Code** in my case). This made the development laborious at some point because I usually had to restart the application quite often.

→ The lack of documentation made it hard to understand new terms or concepts although the developers and managers from NVIDIA provided several tutorials and talks about specific features of Omniverse. The code of the extensions was also usually accessible to the users when not obfuscated. On the other hand, NVIDIA have opened a Discord server in order to gather questions, discussions and announces in one place but the community is also quite new which made it difficult to find help or advice. Finally, visual effects such as Fire or Rain represent only a tiny part of what Omniverse can actually offer to its users therefore few people are working on it inside the community.

→ One last issue is that the platform is still going under breaking/important changes by NVIDIA developers. This could create issues with the solutions I implemented after my departure as some methods I used could be deprecated after a major update.

C Existing and chosen solutions

C - a Fire effect

Omniverse has an extension called **Flow** which has been specifically made to render advanced realistic fluid simulation, such as fire, smoke or cloud. The rendering is basically done by projecting a ray on a 3D grid made of unit cubes named *voxels*. The extension also provides a collection of presets which are pretty much the same object with different parameters that controls the color, volume of the fluid.

Each preset is the combination of four **Flow primitives** (which is a type of USD primitives introduced by Flow) that matches one specific purpose:

→ The **Emitter primitive** which can be of type Sphere, Point, or Box and that includes all fire metrics such as fuel, burn, smoke, temperature amount and so on. According to the type of emitter chosen, the primitive will also have additional properties, for example a radius attribute for the sphere case.

→ The **Simulation primitive** controls parameters related to the behavior of the fire regarding other objects in the scene, for the density of the grid cells, the time scale of the fire -which helps creating a slow motion effect-, the collision properties of the fire...

→ The **Offscreen primitive** contains all properties related to the appearance of the fire such as the color of the fire and the shadows.

→ The **Render primitive** is the least obvious as it possesses parameters to directly control the rendering algorithm of the fire. It also contains properties to debug the simulation such as the wire-frame mode that displays the cells (also called *blocks*) used to render the effect.

Figure 4: *Flow presets, allowing user to directly render smoke, fire or dust in the scene.*

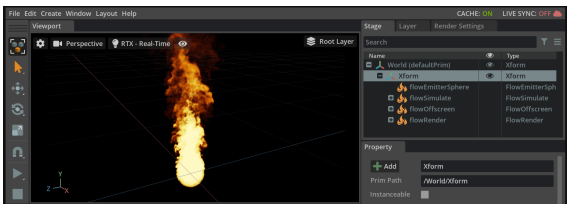


Figure 5: *Flow primitives.*

As I did not want to reinvent the wheel and because I would not have had time to implement my own fluid simulation renderer amongst all my other tasks, I decided to use this extension. This decision solved partly the first and second points of the work I had to do as the appearance of the fire and smoke are quite realistic. However, having only a source of fire by itself in a 3D scene does not add much realism, what we do want is to have objects reacting when subjected to the fire: we want *interactions*. I also had to simplify the attributes of the preset because Flow possesses a lot of settings which can be sometimes technical or not easy to understand.

C - b Rain effect

Rain is an effect which is not as important as fire but that still adds a good amount of realism to a scene. As discussed with the team, the minimum required to make a believable/realistic rain is to render at least the drops falling from the sky and their effect on the ground. Other effects such as wind could come later.

There is no existing example in the community of rain effect, or any particle effect for that matter. I actually spent a substantial amount of time searching for extensions, testing and adapting them to only found out at the end a feature missing to implement the complete effect.

→ The **Particles extension** based on *Omnigraph* is the first extension I considered. It enables the user to build particle system using various Omnigraph nodes that have their own purpose: emitter node (in charge of spawning particles), solver node (in charge of updating the particles), collision detection (which kills particles when colliding with another mesh in the scene) and so on. Although these features could have helped me a lot, there are many reasons why I did not use this extension. First of all, it was still going under major changes during my internship, which would have made my solution most certainly unusable after my departure. Secondly, this extension is pre-compiled from C++ which makes the source code unavailable (obfuscated) for the users. This point is a major issue as we cannot adapt the provided features to Synset needs if necessary, which was my case.

→ The **PhysX extension** is the Omniverse extension that adds true-to-reality physics to 3D scenes ranging from simple collision to vehicle dynamics, fluid dynamics or blast destruction. More specifically, this extension (which contains the Flow extension) provides a connection be-

tween USD physics content and the NVIDIA PhysX simulation engine.

→ The **Warp extension** helps the user write Omnigraph nodes that use an NVIDIA python library called Warp. This particular library converts python code to C++/Cuda just in time, allowing developers to write GPU-accelerated code. Warp already contains model simulators to represent particle systems, but the same issue happen with this solution as with the first extension: the source code was not modifiable and therefore there was no way to add additional behavior to our particle system. Therefore, the solution I came up with was to build my own particle system from scratch using Warp and Omnigraph as it was the only way to meet the two minimum requirements specified at the beginning of this section.

IV Research and development

A Work environment

A - a Development Application

During my internship I only used the Omniverse Code application which was launched in early 2022 and serves as an environment specifically designed for developers. Visually speaking, Code looks very much like any 3D Engine we can find elsewhere with a blank 3D scene we can add objects and effects to (see first Appendix picture). However, it also provides an editor allowing users to add embedded python/C++ code. As said before, it also possible to add or remove custom-made or built-in extensions inside Code, which is, as a reminder, a Kit application just like Synset. Extensions in Ommniverse can be written in C++ or python but as Synset is coded in python, I mainly coded in python. Some parts of my solution also contains code written in HLSL which is a language designed to write shaders.

A - b Collaboration tools

Each team in the CVLab uses an application called **Jira** (made by an external company named Atlassian) for all managerial purposes. This tool allows groups to keep track of issues or tasks over time. More specifically, Synset works in Agile method and uses a Kanban board to keep track of every member's progression in the group. The team also has daily meetings in which each member is supposed to explain briefly the work that has been done the day before and the tasks that one is going to tackle the current day. The team also has weekly every and monthly meetings in order to discuss long-term goals and achievements for the whole team.

The CVLab also uses **Confluence** (another Atlassian application) to store all documentations and other useful contents for colleagues and managers to see.

A - c Daily organization

I used my personal tablet to keep track of my progress and organize myself in more details. At the end of each day, I used to wrote what tasks I had to tackle the following day and each task implemented and tested was highlighted in red. This way, when arriving at the office each morning I already had all things planned for day. It also helped me to show my progress or discuss potential issues with members of my team during the daily meetings.

B Fire effect

B - a Implementation on the Synset app

Fire sources (Fire Smart Objects)

A scene in which we can observe burning objects is in reality made of two components: a fire source, which role is to initiate the fire and other objects which can interact with the source. There are two cases to consider when speaking of fire interactions as each object is either able to burn easily and has the ability to spread fire (**inflammable object**) or is incapable of igniting or burning when subjected to fire (**nonflammable object**).

In practice, the fire source will be defined as a Smart Object of type *Fire* and the other interactive objects will be defined as Smart Objects of any type different than Fire. It is important to note that a Smart Object of type Fire can only be created on one of the two custom-made presets stored in the Synset repository on Nucleus. Each preset represent a source of fire with a different geometry: Cube or Sphere.

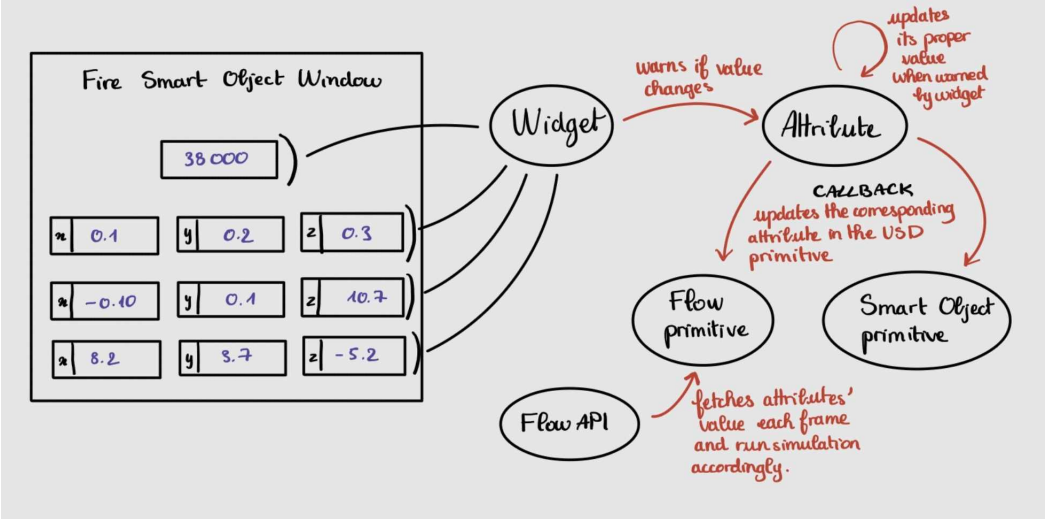


Figure 6: *Illustration of the UI's operating regarding the Fire Smart Object. The Smart Object primitive refers to the USD primitive that has been provided with additional Smart Object properties.*

There are several steps that led to the full implementation of the Fire Smart Object. In the first place, I had to understand how Synset UI worked and how to communicate with the Flow extension. These two steps are described in the figure above.

Within Synset UI, each widget is bound to an attribute of the same type. This dependency allows any window containing the widget to update itself when changing the value of the attribute. Each attribute is also linked to the Smart Object primitive it belongs to which means it can read/write from/to the USD file at the given time. Each attribute's callback can also be authored to implement specific actions when the value of the attribute changes.

In the case of the fire source, we want to be able to connect Synset with the Flow API so that the changes applied to the Smart Object are directly taken into account by Flow fluid simulation. One way to inter-connect the two extensions is to make use of the USD file that contains the current scene with all defined primtives. As Flow primitives' properties are accessible via the USD API of Omniverse, we can override the default attributes' value of each prim with that of our Smart Object prim, as seen in the figure above. Therefore, the attributes controlling the fire source need to be linked to their exact corresponding Flow attribute inside the fire source. In practice, this means that these Smart Object attributes ought to have the same USD name and

type as the Flow equivalent. I also had to store the full USD path to the Flow prim in the stage so that each time the value of the Smart Object attribute is changed, I could also change the value inside the Flow attribute within the callback.

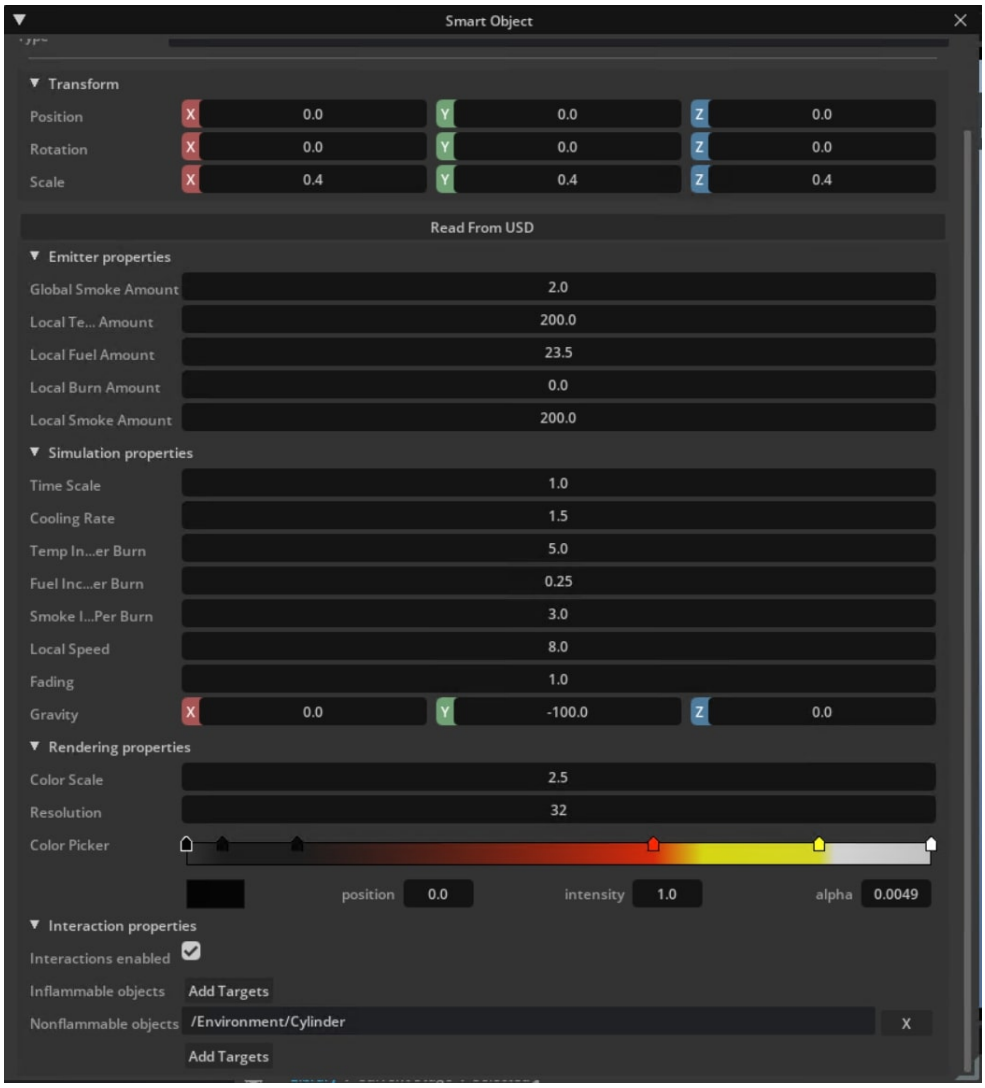


Figure 7: *Fire Smart Object window. Note: a field displaying the path, the name and the type of the Smart Object prim is missing on the figure but it is visible in the next figure.*

The first 3 categories of the Fire properties are also present in the Flow primitives but as said before, only the necessary ones are kept inside the Fire Smart Object. Once I knew what Flow attributes to keep based on their impact on the simulation and how to define their proper callback I could start creating the corresponding widgets and attributes.

When I started to create the widgets and attributes, I already had the majority of them at my disposal except for the widget that controls the color of the fire. This particular widget corresponds to the color picker we can see on the figure below. I managed to exactly reproduce the widget as it is in the Flow UI by using the source code of the extension and by re-programming some of the functions that were unavailable to the users (pre-compiled from C++).

The color picker widget communicates with an attribute (called *ColorMap* attribute) defined as a list of positions and a list of values both represented as a single 'key' widget over the gradient. Each value relates to a 4D vector of floats giving the color in RGBA code and an intensity float which is bounded between 0.0 and 10.0 corresponding to a multiplication factor for the color. The values and positions can be changed with the widgets below the gradient, or by sliding the

'keys' along the gradient. The **ColorMap** lists of values and positions are then written into the Flow primitive and Fire Smart Object primitive at the same time, just like every other attribute.

The methods made unavailable to the users were mostly related to the reconstruction of the gradient of colors present in the widget. In order to do that, I used the key positions and key values of the color picker attribute and used linear interpolation to obtain the gradient of colors.

Inflammable/nonflammable objects (non-Fire Smart Objects)

Once the Fire Smart Object was created and functional, the Synset Application could already render a source of fire in scene along with a window where the user could modify its appearance, position, and other properties. The only thing incomplete was the interactions between the fire source and other objects in the scene. From the Fire Smart Object perspective, I simply added two lists of USD paths (*Relationships*) to store the references of the inflammable and nonflammable objects linked to the given fire source. The fire source also contains a boolean attribute that indicates whether we want the source to be interactive or not.

After defining the *Interaction properties* inside the Fire Smart Object, I moved on to that of all Smart Object types different than Fire. I defined a common section also called *Interaction properties* which contains attributes allowing the user to control the reactivity of the object regarding the fire source (Fire Smart Object) it is being linked to.

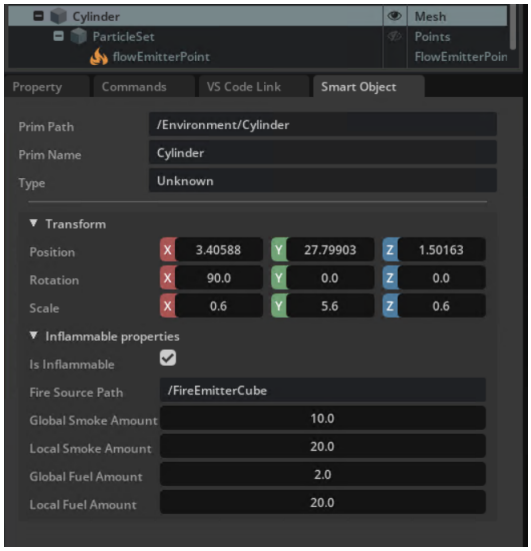


Figure 8: *UI of a Smart Object (type other than Fire) when it is marked as inflammable.*

→ The first (boolean) attribute indicates whether the Smart Object is an inflammable object or not. If not authored, the object is considered as nonflammable.

→ The second (string) attribute contains the path to the Fire Smart Object (fire source) linked to the current object. Its value is empty at creation. It also becomes empty when the user remove the current object from the list of inflammable/nonflammable objects in the corresponding fire source window.

→ The four following attributes are basically the same thing as in the Fire Smart Object case: Flow parameters that will be overridden in the corresponding Flow primitive. These properties enable the user to control the reactivity of the object regarding the source of fire it has been linked to.

When arriving at this stage, we have all the UI defined in Synset for the interactions part and we just have to define the proper callbacks to make objects actually interact with each other. In practice, when adding/removing an object to the interactive objects lists inside the fire source, there are only two cases to consider -apart from the error cases which will be listed later- regarding the nature of the object itself: nonflammable or inflammable.

Nonflammable objects callback

→ The nonflammable case is the only part that didn't require any coding. It was also pretty easy to find out after reading Flow documentation as the extension already provides tutorials on how to obtain the collision effect between the fluid and a mesh in the scene. The first step to make the collision happen is to add a PhysX collider to the primitive which will allow our object to collide with other objects in the scene. In practice, as collisions can become really expensive given the size and nature of the mesh we ought to simplify the mesh as best as we can. As a result, PhysX extension provides several approximation methods for meshes.

In the case of Flow, the documentation requires the mesh to be approximated via the Convex Hull which is the only method that allows to have a bounding shape that is closer to the real object's mesh. The final step is to enable the collision in the Flow primitive by simpling setting an attribute named physxCollisionsEnabled (boolean) to true. When we remove the object from the list, we can simply set the approximation mode in the mesh to *none* and all collisions will be disabled.

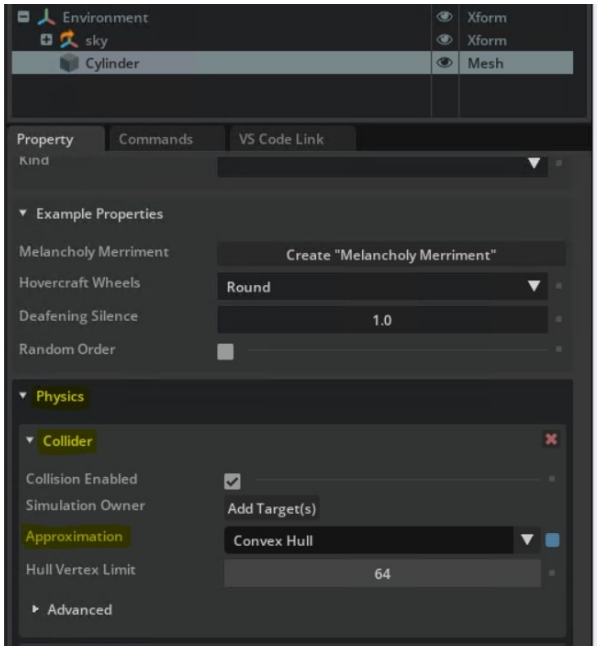


Figure 9: *Physics Collider properties in yellow.*

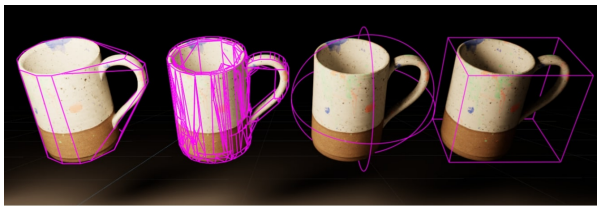


Figure 10: *Mesh approximation methods implemented by the PhysX extension.* ***1.*** *Convex hull* ***2.*** *Convex decomposition* ***3.*** *Bounding sphere* ***4.*** *Bounding box*

Inflammable objects callback

→ The inflammable case needed much more research as the documentation or the tutorials provide no clue about how we can make a random mesh react to a source of fire.

The idea behind the solution is that Flow primitives don't possess any property to 'burn' primitives from other extensions in the stage. However, Flow emitter primitives do have an attribute called *layer* which is an ID (an integer) that helps separate/regroup the simulation of the different sources of fire in the stage. For example, Flow primitives being on different layers won't

have any effect on each other whereas the interaction are enabled if they are on the same layer.

From this assessment, we understand that objects we want to burn needs to be linked in some way to a Flow emitter primitive. Given that an object can have any shape possible, we cannot use emitters of type Sphere or Box. However, Flow has a special emitter called Flow Emitter Point, which binds Flow parameters to a point cloud. Putting this point emitter on the same layer will allow it to interact with another emitter which will correspond to the fire source. Therefore, the steps to obtain an inflammable object are:

→ **Step 1:** Creating a point cloud from the current object mesh. Point clouds are a specific type of primitive in USD, which is called GeomPoints and are initialized based on a list of points and widths. To create the point cloud from the mesh, I used the Meshlab python module which contains wrappers to the Meshlab API. However, USD meshes can be represented by triangles or quadrilaterals whereas Meshlab only uses mesh made of triangles, so I first had to triangulate the provided mesh. This means retrieving the list of vertices' positions, indices and the list which provides the number of vertices in each face of the mesh in order to transform each quad I encountered in the list as a set of two triangles.

After this step, I was able to initialize a Meshlab mesh and run a sampling algorithm based on Poisson distribution that outputs a list of 3D positions. These positions basically represents the mesh as an uniform point cloud. I also used the volume of the mesh's bounding box as a way to adapt the number of points and their width to the size of the mesh.

→ **Step 2:** Once I had the point cloud, I created a FlowEmitterPoint primitive that I further linked to the GeomPoints primitive. This operation allows each point in the cloud to be considered as a local source of fire, which parameters are that of the FlowEmitterPoint Prim.

→ **Step 3:** After that, there are several properties to modify in order to make the point cloud behave like an inflammable object. First, putting all temperature related attributes to 0.0 is necessary as these parameters make the fire fade over time, which is an effect that we don't want in Synset because we need to represent huge and lasting fires in public places just as in reality. Finally, the layer of the newly created Emitter Point needs to be on the same layer level as the fire source because we want them to interact together.

When all these steps are done, we can control the amount of fuel and smoke of the inflammable object by modifying the four related properties in the Smart Object Window. These properties, when modified, will update the FlowEmitterPoint primitive just like it does with the Fire Smart Object case. When removing the object from the list of inflammable objects in the list, we simply delete the previously created (FlowEmitterPoint and GeomPoints) prims and clean the path to the source inside the given object UI.

Figure 11: *Scene with a Fire Smart object bound to a cube preset and an Unknown Smart Object bound to a cylinder marked as nonflammable.*

Figure 12: *Scene with a Fire Smart Object bound to a cube preset and an Unknown Smart Object bound to a cylinder marked as inflammable. The point cloud is made invisible in the scene so that we only see the material of the mesh burning.*

Error cases

Below stands the list of all the user cases that raise an error when using the fire related features within the Synset app. These corner cases are necessary to prevent any behavior that might break the application's operating.

- ✗ User cannot add a primitive that is not defined as a Smart Object.
- ✗ User cannot add a Smart Object that is already bound to another Fire Smart Object.
- ✗ User cannot add a Smart Object to a Fire Smart Object that hasn't its intercative properties enabled.
- ✗ User cannot mark a Smart Object as inflammable if the primitive it's being bound to doesn't possess a mesh. (*This would cause the point cloud generation to fail*)
- ✗ User cannot add a Smart Object that don't belong to the list it's being added to (cases: adding an object marked as inflammable to the nonflammable objects list and vice-versa)

B - b Tests and optimization

Writing unitary, integrated or end-to-end tests is usually not that relevant in the case of 3D simulations. However, as for the Synset implementation, I didn't have time to implement such tests in a script that could be run for example to verify that the error cases listed above are correctly handled. Instead, I just tested directly the error cases in a 3D stage by looking at the console outputs in each case. To make sure each attribute worked as expected, I usually tested each newly added attribute by opening a new stage in Omniverse Code with one of the two fire presets (cube/sphere presets) and looking at the USD properties of each primitive in the scene.

As for the rendering part of the fire, the results of a newly implemented functionality are generally seen directly when looking at the scene and moving the camera. However, I did put some scenes of reference on Nucleus to test the Fire in Omniverse in different cases, which could count as "tests" for this part.

Regarding optimization, one problem with the Flow extension is that the amount of GPU memory necessary to render all fire cells/blocks in the scene can skyrocket fairly quickly when adding inflammable objects to the scene. Indeed, the grid in which the fire is rendered is responsible for most of the memory allocated by Flow. This means that its parameters tend to have the largest impact on performance and memory consumption. In practice, small grid cells will

produce more details in the fire simulation but at higher computational costs so the only way to balance the memory usage for external users is to increase the cells' size.

C Rain effect

C - a Features

Particles systems are extremely common in computer graphics as they can be used to describe a large variety of effects ranging from snow, rain, sandstorms to eruptions or even highly detailed water. They usually function by emitting new particles each frame and updating all newly created and previous particles according to a set of physics forces or rules which can cover only the barely basic interaction such as gravity, to much more complicated ones such as friction, absorption, boucing and so on.

The Rain particle system in Synset is not that different from a 'regular' particle system in the sense that it consists of two central nodes: the **Emitter Node** and the **Solver Node**, except that it is integrated within Omniverse environment. The features of the rain are the following:

- ✓ It is made of two distinct particle types: the **drops** that fall from the sky with a lifespan and a speed defined by the user, and the **ripples** that appear whenever a drop touches the ground.

- ✓ The drops are always displayed in the active camera's field of view and are moving *with* the camera. On the contrary, the ripples only rely on the collision point between the drop and the ground and do not move with the camera.

- ✓ The orientation of the ripples is adapted to the orientation of the ground, which means that the ground can have any wanted shape.

- ✓ Each particle is bound to a material with reflection/refraction properties to mimic water properties.

- ✓ The drops and ripples are re-scaled according to their distance to the camera's position. The reason for this is that particles far from the camera are almost invisible to the screen, which means that putting a huge number of particles wont make the rain much thicker/denser. The only way to not waste the amount of particles was to re-scale them a tiny bit so that fewer particles are needed in order to produce a dense rain.

- ✓ Each ripple's mesh is enlarged and flattened when appearing on the ground to give the effect of water spreading on the ground.

Figure 13: *Rendering of the rain within the scene of reference, with a lighting cube and a deformed sphere as a ground.*

From the figure below, we can see the reflection properties of the drops near the lighting cube and that of the ripples on the ground. The drops are almost invisible but it is precisely because we don't usually notice rain outside when looking at the our surroundings (sky, buildings...). Instead, we generally stare at the ground to see the ripples and deduce the current rain from it. We can also notice how the ripples stick to the ground even though it is a deformed sphere.

The rain system is one of the most interesting parts of my project. However, as it has been made completely from scratch there is not enough space to explain properly every advanced computer graphics and USD notions used inside the code. Therefore, I invite you to read the full documentation I made for my co-workers withing Synset if you are interested. Be aware that it has been written for people who already have at least basic knowledge in USD, 3D computer graphics and Omniverse.

Contrary to what has been said with the fire, the Rain Smart Object was quite easy to implement as it contains the main properties of the rain (maximum particles count, spawning rate and lifetime of the particles) and a path that can be authored to add the ground primitive in the scene.

C - b Tests and optimization

As in the Fire case, I created a scene of reference in Synset in order to test the rendering and the features of the rain. I also created a rain preset on Nucleus to test the Rain Smart Object. Omnigraph also generates automatically tests to verify inputs/outputs types in the new created nodes.

Regarding memory optimization, the Rain effect is not that costly regarding GPU memory as all the buffers of the particle system are stored on CPU.

The main optimization for the simulation's frame-rate was to spawn particles directly inside the field of view of the camera. Indeed, rendering rain (or any object for that matter) outside of the camera's field of view is definitely a waste of performance as users are not able to see anything outside this area. However, this optimization was not available in Omniverse during my internship so I had to find a solution that allowed me to filter the particles from the beginning, meaning at the time of emission.

At the end of the internship, the frame-rate of the scene of reference with a spherical ground and around 4000 particles in total ended slightly above 30 frames per second which is sufficient

but not that great if we consider the high performance GPU I used for rendering: the Nvidia L4. However, there are several reasons that could explain this result, some of them being:

→ Retrieving/Sending data from the CPU to the GPU and inversely is known to be a very costly operation (time wise), that is why particle systems are nowadays made entirely on GPU. However, in order to update the USD primitives in the stage each buffers ought to be sent back and forth to the CPU in order to use the USD API, which makes these passes inevitable.

→ I also highly suspect that the particle system is executed more than one time per frame but I didn't have time to search more on this subject.

V Analysis

Overall, the goals of my project in terms of functionalities were reached at the end of my internship as I succeeded in creating Smart Object of both Rain and Fire types with the main properties to control the rendering of each effect and the corresponding interactions.

A Fire effect

The Fire Smart Object is able to handle all interactions with other objects in the scene and possesses two presets (one based on a cube, one based on a sphere) saved on Nucleus. The fire generated by the Smart Object can be fully customized regarding the color, the intensity of the flames and so on. The only feature I would have liked to add is wind for which I had a solution with Omniverse but did not have time to implement in the Synset application.

There are also some features missing regarding the optimization part. For instance, the application does not automatically adjust the density of the cells regarding the amount of memory usage which means that it is up to the user to do it.

B Rain effect

The Rain system was complete as long as there were drops falling from the sky and ripples forming where the previous particles had touched the ground. The corresponding Smart Object also presented all necessary attributes to control the rain.

Though the idea here was not to render the most realistic rain of all because I had limited time on this part of my project, I do believe there are many ways to improve the rain in Synset. All of them are listed at the end of the rain documentation, in the Appendix.

Also, searching for solutions, optimization or improvement takes a good amount of time as each idea typically needs implementation, testing, tweaking and so on. It also means that at some point I had to stop improving the rain effect after a few weeks working on it because I would not have had time to implement both Rain and Fire Smart Objects in Synset by the end of my internship.

Finally, I do think that I could have gained in efficiency if I was more organized regarding all the features I had to implement in order to make the rain system work. Indeed, there are cases where I happened to find a new bug or issue in the simulation and dropped immediately my current task in order to correct it, instead of writing it down and coming back on it later. This haste would usually make me forget where I left my previous work which, in the long run, results in an non-negligible waste of time.

VI Personal review

Overall, this internship at Eviden was a fulfilling experience both on a human and professional levels.

Firstly, this 15 weeks-long journey happened to be my very first professional experience on a computer science related topic. At the beginning, I was anxious about working in a completely new environment that might be stressful or where employees put too much pressure on others or themselves. I was also afraid to fail meeting my team's expectations. But these fears went away the moment I entered the Computer Vision Lab and felt the goodwill and sympathy of the people working there. During this internship, not only I did gain more technical knowledge but I also learned so much from the people I was seeing everyday at work. By talking and sharing experiences with everyone around me I developed a new outlook on the corporate world and I saw how people can be happy, content and proud of their job.

I am really grateful for knowing about all the engineers, managers and interns from the Computer Vision Lab with whom I shared many hilarious conversations or board games sessions at lunch time. I used to deem from my relatives that work was not an appropriate place to develop fulfilling relationships and share good times but from this experience, I do believe that it is possible. In my case, building good relations with my co-workers made every difficulty easier to apprehend as I was confident in sharing my problems with others.

Regarding the topic of my internship, I really enjoyed working on the Omniverse ecosystem. I did struggle the first week to familiarize myself with all the vocabulary and different concepts that come with it such as USD, the Omniverse extensions or even with the Synset application. But overall, this experience was a real opportunity to discover new topics in computer graphics. It broaden my outlook on this field as I got to explore and discover more tools to simulate 3D scenes with the use of Omniverse *extensions*. While exploring new concepts was exciting, I also really enjoyed incorporating my current knowledge into concrete solutions as for the Rain particle system, for instance.

Finally, given that my topic was fairly oriented towards research, I started to develop various skills that will certainly help me in my future carrier in computer graphics. On one hand, it encouraged me to be as curious as possible as I had to search for all existing solutions to do the effect I wanted. But most importantly, while persisting to make a new solution work might be worth it, this experience taught me first and foremost to be patient and to assess correctly the different solutions I had in mind, in order to anticipate whether some might fail or succeed. For example, when working on the Rain particle system, I spent around one to two weeks testing an extension that did not help me to do the effect at the end. From that mistake I learned to dig more thoroughly into each solution's specifications before testing it. I also learned to organize myself better as to when one feature involved a lot of underlying steps that each needed to be dealt with in order and with patience.

On a more practical note, being within Synset team and working closely to Nvidia got me realizing about all the jobs we can find in the computer graphics field. I find these information quite interesting as I had no idea what jobs could be provided within corporations in that area.

VII Conclusion

During my internship, I joined a R&D team called Synset that builds an application to generate large and realistic synthetic data-sets. While acquiring real data can be a hassle, these computer generated data-sets are used as additional train/test sets for AI models aiming to detect incidents in public places. My goal inside the team was to add visual effect to the scenes inside the Synset

application, focusing on fire which is used to train fire detection models, and rain, which is used to add more realism.

This project covered a wide range of tasks within Nvidia Omniverse's platform, including: exploring as much Omniverse extensions as I could to solve my problem, helping my co-workers build a UI with custom-made callbacks, adding new features to the Synset application or simulating a particle system from scratch.

At the end of my internship, all the main goals of my project were reached. While I succeeded in integrating functional fire and rain to the Synset application, there were still improvements that could be made for both effects. These enhancements, related to optimization and realism, were mainly concerning the rain effect for which I had to create the entire simulation from scratch in limited time.

As a conclusion, it was a great pleasure to work inside the Synset team not only because I quite enjoyed learning and using plenty of new concepts in computer graphics, but also because I undoubtedly appreciated the people with whom I shared these 15 weeks. As I prepare to finish my last year of school at Ensimag, I foresee pursuing my learning journey in computer graphics by taking more courses related to this vast field.

VIII Appendix

A Pictures

A - a USD stage

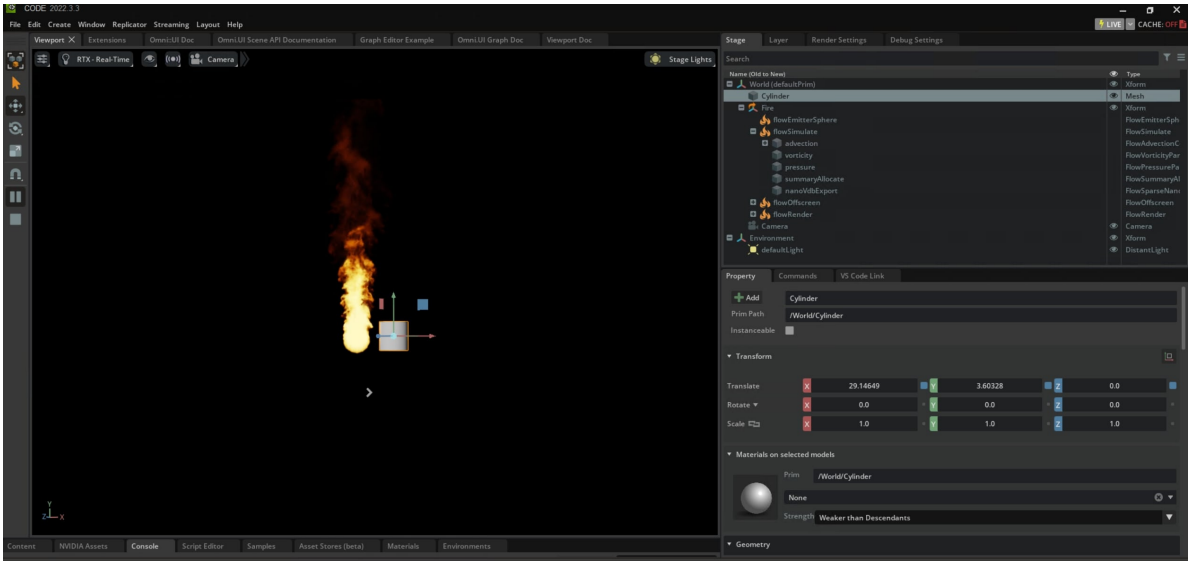


Figure 14: *An example of stage in Omniverse Code.*

A - b Fire Smart Object

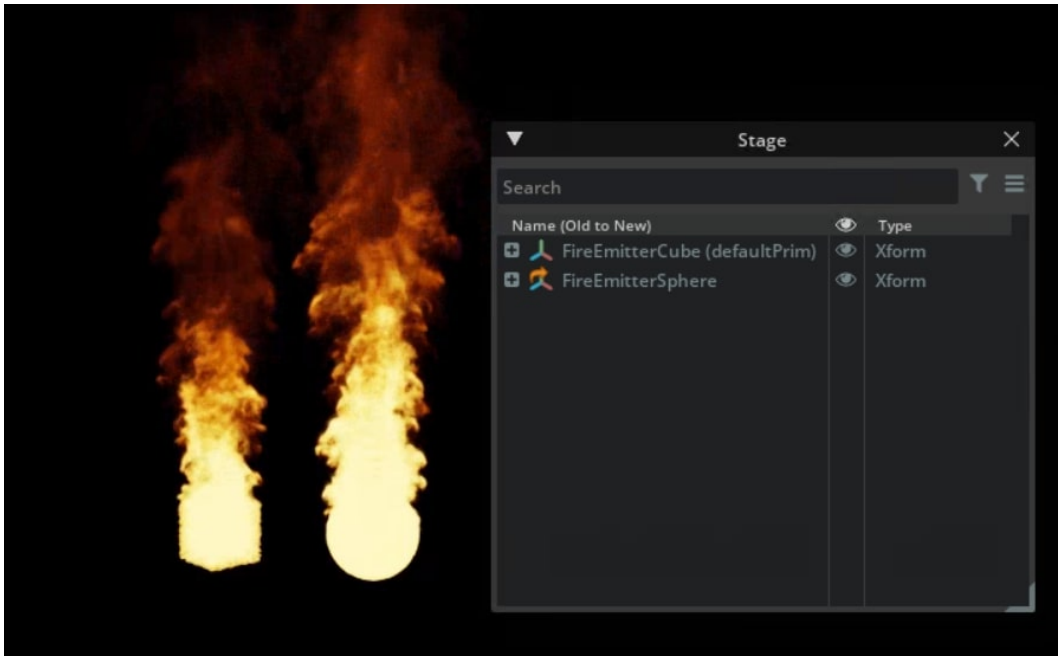


Figure 15: *The two fire Smart Objects presets: based on a cube source (right one) or on a sphere source (left one). Each source can be deformed do to other shapes. Note: the sphere preset primitive has a different logo because it is not defined stage in the stage, it just references another USD file which contains the preset's definition.*

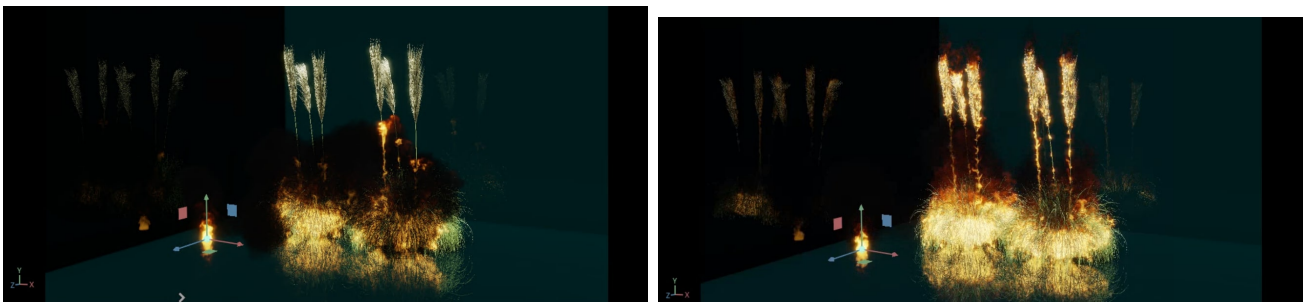


Figure 16: *Source of fire inflamming plants at different timestamps. Scene of reference for the fire feature in Synset.*

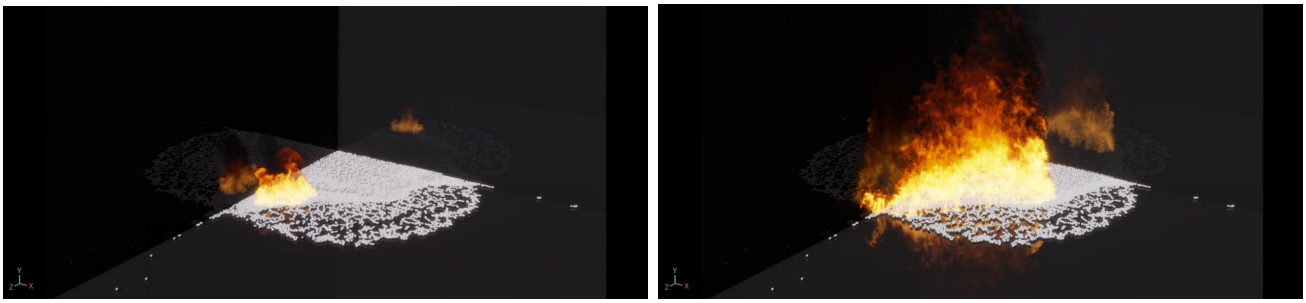


Figure 17: *Source of fire inflamming particles at different timestamps. Scene of reference for the fire feature in Synset.*

Figure 18: *Scene of reference when the fire is exposed to external wind forces.*

A - c Rain Smart Object

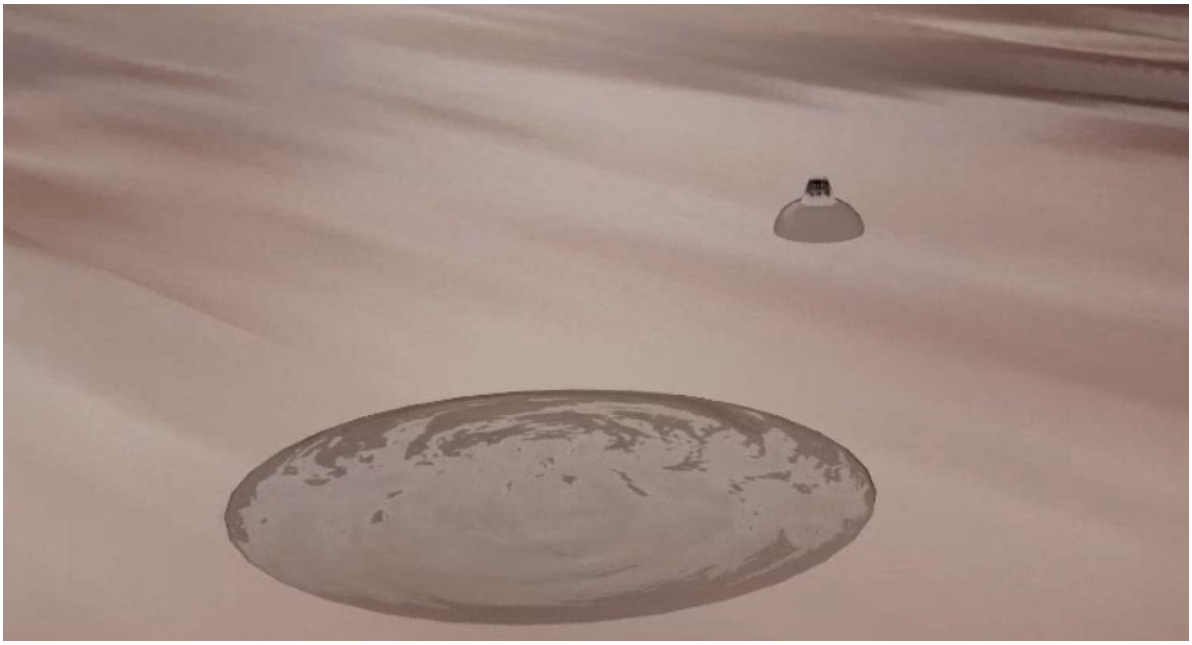


Figure 19: *Spreading of the ripples on the ground at different stages of the particle's life (front: old particle, back: young particle (but not newborn)*

A - d Personal organization file

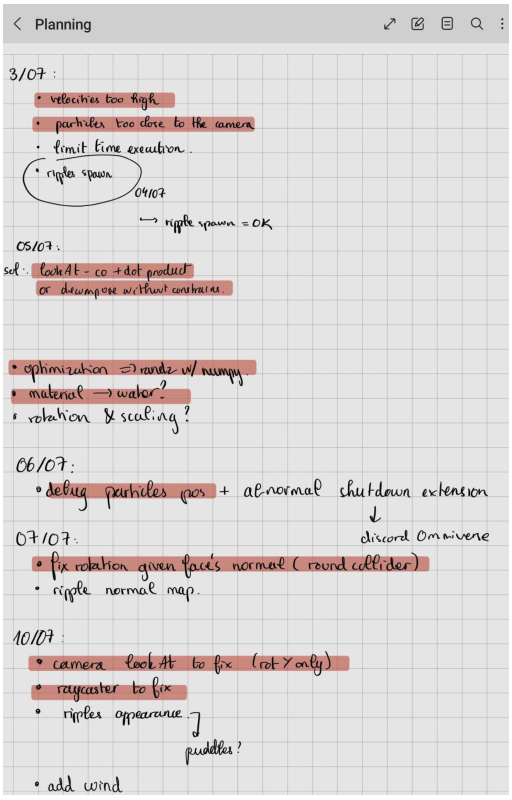


Figure 20: *Example of a page in my tracking file for the internship.*

B Glossary

- **Rendering**: drawing onto the screen.
- **Rendering pipeline**: sequence of steps that leads to the display of an object on the screen.
- **Shader**: program that runs on GPU. Shaders are usually used to render effects on the screen but they can also be used to do GPU-accelerated computations.
- **3D Asset**: refers to any type of 3D contents that can be stored digitally, ranging from simple mesh objects to complex 3D scenes.
- **Preset**: special type of asset that is used to convey identical properties or settings across any type of 3D scenes.
- **Mesh**: 3D (or 2D) points network.
- **Xform**: special type of USD primitive that only contains translation, rotation and scaling 3D vectors. This type of primitive is used to apply transformation to objects in the scene.
- **Material**: in Omniverse's context, refers to a texture that is applied to an object in the scene. For example, a wood material or a water material.
- **Widget**: visual representation of a UI (User Interface) element. For example, a 3D vector widget in Omniverse will typically display three blocks, one for each component of the vector.
- **Callback**: in Omniverse's context, a callback is a function bound to an attribute that is called every time the attribute's value changes.
- **Collider (Omniverse PhysX extension)**: additional behavior that can be applied to a mesh primitive which is from then on able to collide with other objects in the scene.
- **GeomPoints**: type of USD primitive that is specifically used to represent point clouds. It possesses two array attributes respectively referring to the position of the points in the cloud and their width.

C Links

- **Omniverse** overview
- **USD** overview
- **Warp** documentation (library only), **Warp** documentation (within Omniverse)
- **Omnigraph** overview
- **Flow** documentation (library only), **Flow** documentation (within Omniverse)
- **PhysX** documentation (within Omniverse)

D Documentations

Implementation on SynSet - Fire and Smoke

Introduction

- The class "Fire" contains all parameters and information necessary to control a source of fire (=the emitter) and its interaction with other objects in the scene. This type of SmartObject is based on two Nvidia Omniverse extensions : **Flow** (fluid dynamics extension) and **PhysX** us ed to detect collisions between the fire and other objects in the scene, see the parent page for the links.

- Two presets of emitters are present initially : a **sphere** and a **cube**, but these can be translated, rotated and rescaled in 3 dimensions thanks to the transform (Xform) binded to each emitter. This can allow the user to do a lot of different shapes of emitters such as : **disk**, **rectangle**, **el lipsoid**... Each emitter is basically a particle system (generated with physX) binded to a 3d mesh. Each particle behaves as a "Flow Emitter Point" which is an emitter of fire according to the Nvidia Flow extension.

- Each smart object of type "Fire" has 4 categories of parameters: **Emitter properties**, **Simulation properties**, **Rendering properties** and **Inte raction properties**. The first 3 categories can be found on the page of Flow simulation (many initial flow parameters won't be modifiable by the user as they don't have much effect on the simulation).
    - The **interaction properties** are *custom properties* added for SynSet purpose because we want the fire to interact/behave differently according to its surroundings. Each object in the scene that can interact with the fire is **EITHER** an inflammable object (that 'spreads' the fire) **OR** a nonflammable object (that 'collide' with the fire). The idea is to stock the relationships with each surrounding object in two lists (inflammable/nonflammable).

Important note!!!: Fires and Smokes belong to the SAME TYPE of smart objects but with different flow parameters (color, smoke amount, fuel...)!

Smart Object of type Fire

- NOTATION: **UI Name [USD Name]**: synset attribute type => definition

Emitter properties

- **Global Smoke Amount [smoke]**: float => global smoke parameter. Coupled with "coupleRateSmoke" (ie when coupleRateSmoke and smoke have both high values), it can help create a lot of smoke above the source of fire. The color of the smoke can be changed in the Rendering properties.
- **Local Temperature Amount [coupleRateTemperature]:** float => temperature of the fire emitted from each particle.
- **Local Fuel Amount [coupleRateFuel]**: float => fuel of the fire emitted from each particle.
- **Local Burn Amount [coupleRateBurn]**: float => burn of the fire emitted from each particle.
- **Local Smoke Amount [coupleRateSmoke]**: float => smoke of the fire emitted from each particle. Be aware that this parameter is closely related to the Global Smoke Amount parameter, if the global amount is set to 0.0, the local amount won't do much effect on the fire.

Simulation properties

- **Time Scale [timeScale]**: float => scales input time. When the value is lower than 1.0, it creates an effect of slow motion on the fire.
- **Cooling Rate [coolingRate]**: float => amount of temperature that fades per cell (voxel) in the fire. Applied to the fire that emanates from each particle.
- **Temp Increase Per Burn [tempPerBurn]**: float => temperature increase relative to burn amount. Applied to the fire that emanates from each particle.
- **Fuel Increase Per Burn [fuelPerBurn]**: float => fuel consumed relative to burn amount. Lowel values will make a given amount of fuel last longer. Applied to the fire that emanates from each particle.
- **Smoke Increase Per Burn [smokePerBurn]**: float => smoke increase relative to burn amount. Applied to the fire that emanates from each particle.
- **Local Speed [buoyancyPerBurn]**: float => upward force relative to temperature. This parameter, couple with the **gravity** parameter can actually be interpreted as the "speed" of the fire in the direction of propagation.
- **Fade [fade]**: float => fading of the fire.
- **Gravity [gravity]**: vec3f => correspond to the direction of propagation. Higher values for each component (x,y,z) increase the speed of the fire in that direction.

Rendering properties

- **Color Scale [colorScale]**: float => increase/decrease the intermediate colors between the key positions but on the gradient widget.
- **Resolution [resolution]**: int => corresponds to the number of texels used to map the colors.
- **colorMap attributes**: => holds rgba and intensity values for each of the key position in the gradient widget. The intensity is a multiplication factor applied to the color. The alpha channel controls the transparency of the color.

Interaction properties

- **Interactions enabled [physicsCollisionEnabled]**: bool => Indicates if the current source of fire can interact with other objects in the scene or not. If **False**, the objects present in the two lists of inflammables/nonflammables objects won't react when touching the fire in the scene. This parameter has been created for future purposes as we sometimes want to disable/enable all interactions at once instead of having to remove/add all the objects in the lists.

- **Inflammable objects [inflammables]**: multi target relationship / **Nonflammable objects [nonflammables]**: multi target relationship=> these parameters are custom parameters (not based on Flow extension). It allows the user to bind/unbind objects of the scene to the source of fire to allow fluid/rigid body collisions. Object added to the inflammable targets will transfer fuel and therefore burn when touching fire (= flammable). Object added to the nonflammable targets will be considered as nonflammable objects.

Other Smart Objects

- The other smart objects have acquired new fields to control their reaction to the fire in case they're defined as inflammable objects. These attributes include:
    - **Is Inflammable [IsInflammable]**: bool => Indicates if the current smart object is marked as inflammable. **CAUTION**: **If False, the object is automatically set as nonflammable!**
    - **Fire Source Path [fireSourcePath]**: str => if the object is marked as inflammable, this attribute holds the path to the fire source (Smart Object of type Fire) that interact with the current object. There can be only one fire source per inflammable object whereas one flammable object can collide with multiple other fire sources.
    - **Global Smoke Amount, Local Smoke Amount, Global Fuel Amount , Local Fuel Amount**.

Callback Handling - Interaction Properties

(*Note*: SO = Smart Object)

Addition/Removal of Inflammable/Nonflammable objects in Fire SO:

- *Note*: the code of this section is fully documented in the code but here are some explanations on how the addition/deletion of object is handled:
- When adding a smart object to the list of inflammable/nonflammable objects, a checklist of error cases is checked beforehand, see the section **Error Cases** below.
- If the object is added/removed to the list of inflammables:
    - When the prim is added, we put its flow layer on the same level as the flow layer in the fire SO (otherwise both flow objects won't interact with each other).
    - When the prim is removed, we simply put a negative layer (all flow layer are positive ints so this make sur the SO won't be able to interact with the fire SO).
- If the object is added/removed to the list of nonflammables:
    - When the prim is added, we check if the collision approximation is Convex Hull and if not, set it to that approx parameter this mode of approximation is mandatory if we want the mesh to interact with the flow prims inside the Fire SO.
    - When removed, we set the approximation to none.

Creation of an inflammable SO:

- When a SO is marked as inflammable, a callback is called to create the properties that will allow the user to control the reactivity of this object. In order to do that, we firstly create a point cloud (see **Error Cases** below for when this doesn't work) that fits the mesh of the prim by using a Poisson Disk sampling provided by Meshlab. This method of sampling creates a uniform sampling on the mesh, which is necessary in our case as we want the points to be as equal distant to each other as we can in order to spread the fire. We then create a flowEmitterPoint primitive that we link to the point cloud and voilà!

Error Cases

The purpose of this section is to enumerate the cases forbidden to the user and that will return an error in the console without stopping the program:

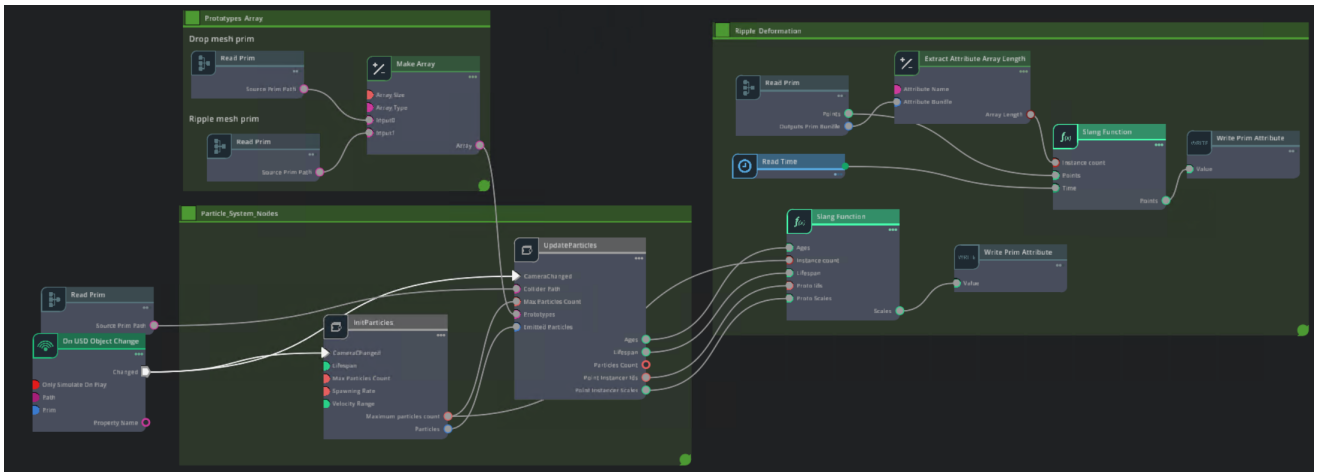
- If the Fire SO doesn't have its interactions enabled when adding a prim to the list of inflammables/nonflammables, the addition will be canceled automatically.
- Only SO can interact with Fire SO.
- Only SO marked as inflammable can be added to the inflammables, same thing for the nonflammable objects.
- SO that are already bound to another Fire SO cannot be added to the list of inflammables of a Fire SO.

One case will raise an exception:

- A SO that doesn't possesses a mesh cannot be marked as inflammable as the creation of a point cloud will be impossible.

Rain Particle System :: Documentation

Illustration & Explanations of the OmniGraph:



- The rain particle system is made of two custom nodes: the emitter node (InitParticles) and the solver node (UpdateParticles).

    - **<u>Emitter Node</u>:** the Emitter Node is in charge of generating each frame a set of particles with the given properties: position, velocity, lifespan and age, which is the percentage of lifespan of the particle. The emitted particles are transmitted to the next node until the maximum particles count is reached. Here are some explanations on the inputs of this node:
        - The active camera primitive is used in order to spawn the particles in its frustum and to scale them according to their depth (distance to camera).
        - The velocity argument serves as a range to generate the random velocities of the particles.
        - The lifespan is the same for all particles.
        - The spawning rate defines the number of particles emitted per frame.
        - The camera properties (transform, angle of view) is updated only when the primitive is changed.

    - **<u>Solver Node</u>:** the Solver node is in charge of updating the particles properties according to their surrounding world. This node is at the center of the simulation, it updates each particle given its type (drop or ripple), its initial metrics (position, velocity...). Here are some explanations on the inputs of this node:
        - The collider node points to a mesh primitive in the stage (or in another file). This primitive can be the result of a merged mesh. If authored, the node will run collisions detection (by perfoming a raycasting) each frame and update particles accordingly.
        - The prototype paths points to several meshes in the stage (or in other files). This allows the user to bind any mesh to each particle in the system (particles can therefore have different meshes even if they come from the same system).
        - The camera properties (transform, angle of view) is updated only when the primitive is change

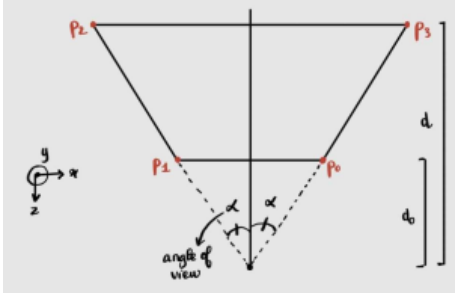
- The outputs of the solver node (UpdateParticles) are then transfered to another set of nodes that will deform the ripples given their age or position in the sphere by enlarging and flatten them with time:
    - In the last version of the rain, one Slang Node is in charge of 'flattening' the points of the initial sphere, and one slang Node is in charge of 'spreading' the points of the sphere (using fractional brownian motion and the percentage of lifetime of the particle).

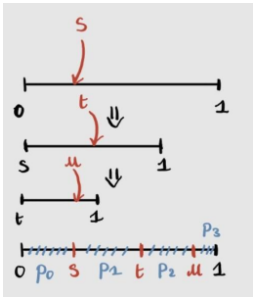
Some technical explanations

- *When launching the stage:*
    - All custom omnigraph nodes (Emitter and Solver) will be bound to a *State* class that will hold intermediate information for each node as the simulation is running. Basically, the Emitter will hold the camera properties (because we don't want to recompute them each frame unless the camera has moved or been changed) and the buffer (position, velocity, scale, age) of the emitted particles that will be overriden each frame. The Solver Node will hold much more information as it has much more tasks to run each frame. The main information is listed here:
        - **Current particles metrics buffers**: position, velocity, initial_scale, age. These are numpy arrays converted to Warp arrays.
        - **Current point instancer buffers**: id, position (prototypes positions) , scale, orientation, angular velocities. These are also numpy arrays that will be converted to Vt (usd array type) arrays in order to be passed on to the point instancer prim. There are 2 position buffers as the current particles positions serves as intermediate information for the prototypes positions. Note that all these buffers (id, scale, orientation, angular velocities) are required by the point instancer prim.
        - **Collider mesh id (in Warp)**. This id is updated whenever the simulation is stopped.

        - **Camera properties** (transform). This transform is updated whenever the current active camera moves.

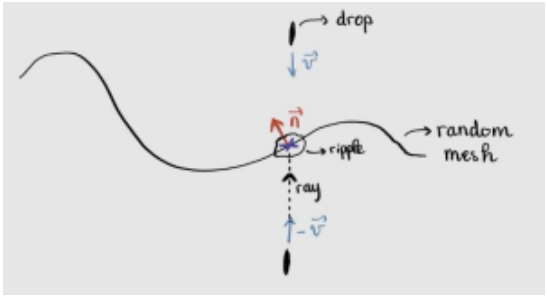
- *When the simulation is stopped:*
    - The emitter node initializes the camera properties (angle of view).
    - All the buffers are initialized with a size equals to the given *maximum particles count* in the **Update Node**. The collider id is set by generating the corresponding Warp mesh. The Solver also initializes the camera properties by creating a new camera if the viewport doesn't have an active camera. Then, the Solver nodes retrieve the transform of the current active camera.

- *When the simulation is running:*
    - The **emitter node** generates new properties for the amount of particles given by the *spawning rate*. More precisely, the age of the particles is randomly chosen around the *lifespan* parameter. The velocity of each particle is chosen between (-v0x, v0x) for the x-coordinate, (-v0y, -0.2) for the y-coordinate and (-v0z, v0z) for the z-coordinate. The position and scale of each particle is a little bit more complex.
        - We know that all new particles are drops, and that these drops need to be generated inside the frustum of the current active camera. The idea here is to generate the positions randomly and uniformly inside a trapezoid which is a little bit above the ground (y~15units), with the depth along the z-axis and centered in 0.
        - 

        - In order to generate a position uniformly in the trapezoid, we use barycentric coordinates that will weight each point. Here is a scheme that shows how to generate these coefficients: we choose the first coefficient randomly between 0.0 and 1.0. The second one is chosen between the first one and 1.0 and so on until there is no coefficient left. By doing this way, we obtain 4 coefficients which values sum up to 1.0 (barycentric coordinates properties). The distance between each coefficient gives the distance to the corresponding point, considering that the first distance belongs to the first point, the second distance to the second point. By setting this order, we give more weight to the first two coefficients on average which translates to the fact that we're having more drops close to the camera. *Note:* in reality, the coefficient **s** is set between 0.0 and a number below 1.0 (between 0.5 and 0.8 for instance) as this coefficient really limits the others.
        - 

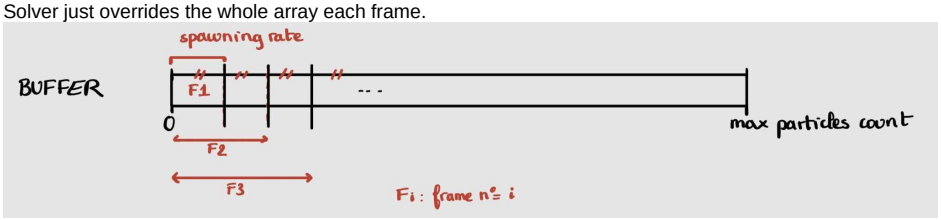
        - The initial scale of each particle depends on its distance to the camera (depth). This feature has been added in order to make the rain more 'dense' which fewer particles. Every drop needs 2 scales float (one for the y-scaling and one for the x and z scalings which are the same given that each drop is symmetric along y-axis). Each ripple needs only 1 scaling attribute (for x,z-scalings) to flatten the initial sphere even though it will be modified further in the simulation. Here, the idea is simply to slightly enlarge and stretch the drops as they are far from the camera.

    - The **solver node** will fetch the emitted particles that come from the previous node. Given these new information, a **Warp Kernel** (which is a function that allows its embedded code to run on numerous threads at the same time) will update the metrics of each newly created particle AND currently existing particles according to its nature (drop or ripple), its position, etc. One important thing to note here is the presence of a collision test, which is necessary to detect the moment where a drop becomes a ripple. This test is done by projecting a ray along the opposite velocity of the particle (see figure below). A warp method called *mesh_query_ray* will perform a raycasting test on the surface of the provided mesh and will return the intersection position if any. This intersection point will be the position of the future ripple.



- The problem is that all ripples are initially aligned with the world (x,z) plane, along the y-axis (which corresponds to the axis of rotation). This is an issue in every case where the collider mesh is not a plane as the ripple won't coincide with the orientation of the mesh, resulting in a non realistic effect. Therefore, the solution is to retrieve the normal to the mesh at the intersection point (red vector in the above figure) which is basically the future axis of rotation of the given ripple. Also, the orientation of the prototypes in the Point Instancer is represented by a quaternion, meaning that our problem is actually to determine the rotation quaternion that sends the initial vector (0.0, 1.0, 0.0) (y-axis) into the normal vector (n).
- When the particle is a drop that doesn't collide with the ground, it looses a small percentage of its age until reaching 2% of its lifetime which case it just respawn at its initial state (position, velocity, etc). When a ripple reaches the end of its life, it respawn as the drop that had given this ripple, at its initial state.

- The Warp Kernel is executed on GPU, which means that the updated buffers need to be brought back on CPU again in order to write into the PointInstancer prim and therefore update the whole simulation. The Solver node updates the numpy /Vt buffers each frame by updating the current existing particles (= the particles created and added into the buffer by all the previous frame) AND the newly created particles that takes an amount of storage equal to the spawning rate inside the buffer. The more the simulation runs, the fullier the buffer becomes until the maximum particles count is reached and the Solver just overrides the whole array each frame.



Important side notes (last update: 25/08/2023):

- Because the particles are stocked inside a buffer with a fixed size, the maximum particles count cannot be changed when the simulation is running. You'll have to pause the simulation first if you want to change it and restart the simulation over.

- **Here are the few features I would have liked to test/implement/correct if I had more time:**
  - ***In terms of features:***
    - The collider needs to be under one single prim mesh, this requires the user to merge the various mesh of the ground in order to make it collide with the rain. I would have rather liked to put an argument that supports multiple meshes.
    - I would have liked to add wind to the simulation. ***[Searched Solution(s):]*** In order to do so, it is necessary to rotate the drops (which corresponds to the prototype id of 0) depending on the value of the wind vector (norm and direction) using quaternions. These quaternions can be set inside the buffer called "instancer_orientations".
    - The drops are currently bound to the camera transform which can be an issue if we're looking above the camera's position in the sky or just below it. This isn't a real problem in 99% of the cases as the drops have a light color and aren't that visible but it's still an issue that hasn't been fixed. The reason for this is that the particles are affected by the 3 rotations (along axis X, Y and Z) of the camera which means that their position is rotating with the camera even though we're just looking at the sky. ***[Searched Solution(s):]*** One solution would be to rotate the particles only in the world (x,z)-plane but I didn't have time to fix properly this issue as the rotation with the Usd format are quite a hassle (each angle of rotation is bound to the interval [-pi/2, pi], meaning we have to reconstruct each angle for each axis and and there are some corner cases I couldn't solve).

  - ***In terms of performance***:
    - I believe there are ways to improve the time necessary to perform one pass (emission & update) in the graph. With ~4000 particles, the framerate falls just above 30fps. This is most certainly due to the fact that not all the simulation runs on the GPU as there are many passes between the GPU and the CPU to update the prim attributes accordingly. ***[Searched Solution(s):]*** I suspect that the existing warp models for the simulation are running entirely on GPU so if it is possible to do the whole collision detection/ripple spawn thing which those models, I think it could be worth taking a look to this solution.
    - It is also highly possible that the whole graph is executed more than one time per frame. ***[Searched Solution(s):]*** In this case, adding a delay could be an option. I didn't have time to make it work with the omni.timeline extension but I think the answer could be there.

- ***In terms of effects:*** The effect of the ripples on the ground could be much improved by:
    - Making the particles accumulate and 'stay' on the ground (but this could hamper the performance of the whole simulation as more particles are needed). Another thing would have been to leave 'traces' of the ripples on the ground's material but I didn't find a solution using the MDL graphs: modifying an existing shader with MDL doesn't seem to be doable but I could be wrong...
    - I would have also liked to have more time to search on alpha blending to make the ripples vanish graduately on the ground. ***[Searched Solution(s):]*** *I tried looking at PrimVars Opacity* attributes but I didn't have enough time to make it work: Omniverse Materials — materials-and-rendering latest documentation (nvidia.com).
    - I would have liked to have more time to improve the spreading of ripples on the ground by adding more displacement (the fractional brownian motion gives values way too high in the Slang Node so I had to clamp them and I didn't have time to find the reason for this...).