Study of Techniques Used in Making Explosion VFX

Nathanaël BROCH
nathanael.broch@yahoo.fr
CNAM-ENJMIN
Angoulême, France

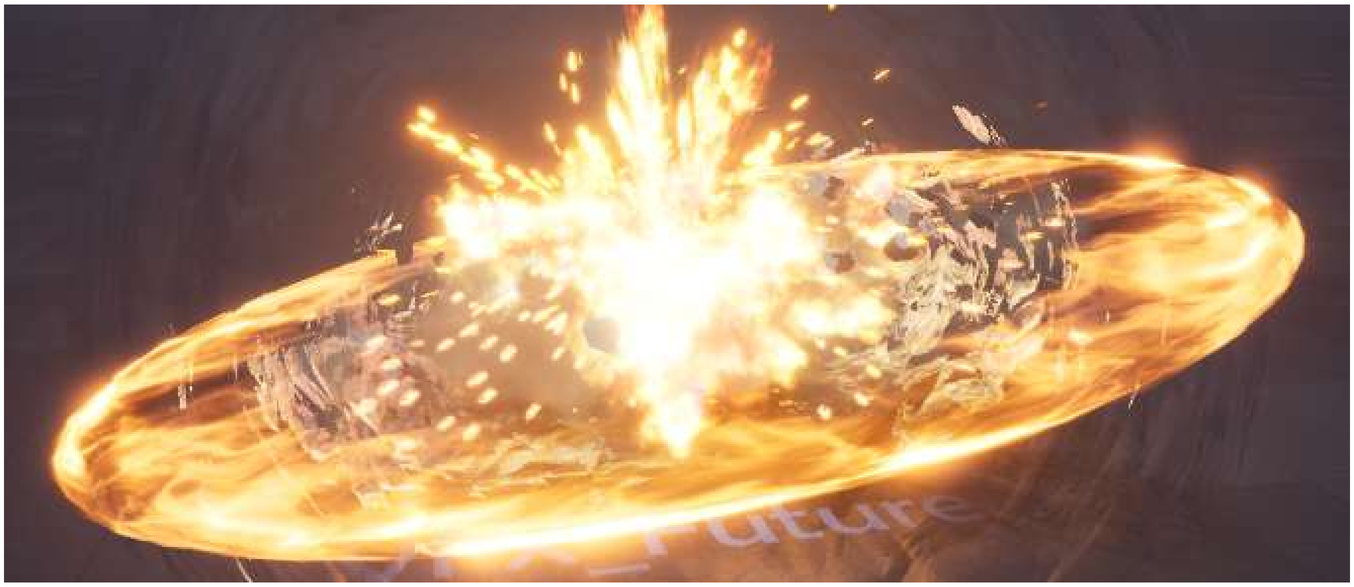

Figure 1: Example of a semi-stylised explosion VFX. Effect by "Future", via Real-Time VFX [16]

ABSTRACT

This report covers the implementation of a custom, real-time explosion VFX in *Unreal Engine*, as well as the making of the assets used in it. Being made from scratch, no default asset from the engine has been used ; as the goal was to learn how to use the other tools and software programs used in VFX making. We will begin with a state of the art regarding explosion VFXs in video games, follow through with a description of the tools used, and then elaborate on the different components of an explosion as well as the techniques employed to implement them.

CCS CONCEPTS

• **Computing methodologies** → **Animation**.

KEYWORDS

VFX, Particle system, Shader, *Unreal Engine*, *Niagara*, Tools

ACM Reference Format:
Nathanaël BROCH. 2023. Study of Techniques Used in Making Explosion VFX. In *Proceedings of CNAM-ENJMIN Experimentation Subject*. ACM, New York, NY, USA, 9 pages.

1 INTRODUCTION

Visual effects, or VFX for short, are tools heavily used in today's world. From filmmaking to video game development, they are almost an unavoidable part of the visual aspect of the media they are used in. There are a multitude of visual effects to use with a plethora of ways for them to be implemented, all to reach the intended approach.

The VFX creation field is a vast one, and standing at the border of it as a novice can be daunting at first sight. In an effort to take my first dip into the world of technical art and effect making, I undertook this research project to learn how to create VFXs from scratch in *Unreal Engine*, a game engine I had never used before. Additionally, I explored new tools in order to learn how to create visual effects from scratch and find ways to dive deeper into the video game development branch.

This report will focus on how I researched about a common effect found in video games : explosions. We will then follow up on how I personally broke down and implemented an explosion.

2 STATE OF THE ART

When bringing up VFXs in video game development, whether we are talking about explosions, fire, smoke, poison, water, magic, electricity or any other things, VFXs are not just present to make things pretty.

Through different artistic principles, how these aspects come together is what separates a good VFX from a bad one, but more importantly, how they manage to integrate themselves into the game's design :

- They understand the **gameplay**. They need to bring clarity to the players by making the game readable first and compelling second. A game needing precision should accurately represent hitboxes by creating areas of focus, making them clear and understandable. A rhythm game's VFXs would highlight the beat of the music, while the ones for a fast-paced game would quickly and accurately signal key information to the player. Visual effects are not just here to make things enjoyable to see, they alert of danger, they reward, they inform : they carry a message to the player. Just by the way they are implemented, they alone can show if something bad, good, or important occurs, whether it is tragic or rewarding. As a result, they are also a great way to contribute to the game's narrative by driving it forward ;
- They use color **values** and contrast at their advantage to emphasize key information. In color theory, a color value is a shade of lightness. Each color comes with value, and brighter colors naturally draw more focus, as well as contrast between different color values : a dark and a bright color next to each other would catch the eye much more than two bright colors. This principle can be used to create areas of focus and differentiate important events from lesser ones ;
- They use **color** palettes and saturation to create a language to the player. In color theory, saturation refers to the intensity of a color. Through the palette and saturation, VFX can follow a theme, and effectively give more accurate information to players that way. For example, the enemy is red and allies are blue ; green is health and electric based attacks are yellow.
- They efficiently use **timing** to improve accuracy, artistic choice and gameplay. Through anticipation, an effect can inform the player that something is about to happen, like a charging spell or an increasingly glowing item, for example. With the same principle, dissipating effects can signal that something happened before. Timing can make effects more impactful as well by expressing power or speed ;
- They use **shapes** to their advantages to convey meaning. Sharp edges are dangerous, round circles are softer. Directional shapes convey the information of motion more accurately, realistic textures can create noise with their details. Shapes are also an important part of their stylization.

All of these artistic principles must be considered when making any VFX, and they in turn affect the technical side of things with stylization.

2.1 stylization

A cartoonish explosion will indeed not be implemented the same way as one that wants to look as photorealistic as possible, and as a result the techniques employed to implement both of these effects in the game can vastly differ (See figure 2). Some might only need to be made with standard particles systems while some others would need an entire simulation or effect that would require specific shaders, tools and even external softwares to be efficiently implemented.

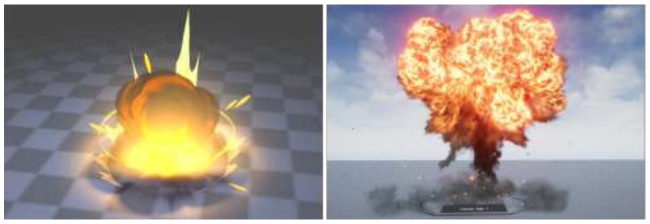

Figure 2: On the left, a stylized explosion, made by "Nobody109", via Real-Time VFX [25]. On the right, a photorealistic nuke, made by "FX Cat UA" via the content "Explosions All" on the *Unreal Engine Store* [23].

However, stylization is not the only thing to take into account when thinking about the different ways to make a specific effect. Other key factors include the machine the game is running on and the context the effect is used in.

2.2 Performance

Performances are the second thing to consider when trying to understand how to implement an effect into a game. Not all machines are created equal, some are more powerful than others, and even the most powerful of machines cannot manage to calculate a heavy amount of simulations together without loss of frame rate. That is not even considering the other operations that the game will run in the background that already use up a lot of memory and processing power.

A cutscene would not need to handle gameplay processes, and more memory space could be used to create more expensive effects. Similarly, visual effects played in real time during gameplay should be more cheap to use, especially if other VFXs can be played in tandem.

Many different tricks can be used to solve this issue while still being able to make believable effects. Machines are made of many different processors, and as an example, one solution could be to delegate the calculations to a different part of the machine. Some effects could also be completely simulated once while being recorded, in order to avoid doing the calculations in real time. From scrolling textures that warp meshes along their paths to faked collisions that trick us into believing water drops actually detect the ground, VFXs can look way more complex than their actual elegant creation process, all while by being easy to run by machines. Making VFXs is not about simulating it, but more so about making it believable enough for the audience.

3 TOOLS USED

The goal of this project is to discover and familiarize myself with *Unreal Engine*, and learn how to make visual effects with it. *Unreal Engine* comes with default textures and materials from the start, and as much as they might simplify the work that needs to be done, I have chosen against using them. My idea was that by recreating them from scratch, I would learn core principles and how to use other tools used in VFX making. As a result, I would have learned not only how to compose effects, but also how to create their base materials ; in turn making me able to create a way bigger array of VFX with the help of my newfound skills. Three different tools appeared frequently during my research to create explosions. I then decided to focus my attention on learning how to use them.

3.1 Niagara

Niagara is *Unreal Engine*'s VFX system, just like how both VFX Graph and *Shuriken* are to Unity. However, if *Shuriken* is for CPU-handled particle systems and *VFX Graph* is for GPU-handled systems on Unity, *Unreal Engine* can make both CPU or GPU particle systems with *Niagara*. It shares some core components with *VFX Graph*, but allows for more functionalities like the ability to create additional functions of our own without programming.

Along with *Niagara*, the second tool we will use to create and adjust visual effects inside *Unreal Engine* are materials. Materials can be opened up to work as a shader, with an interface comparable to Shader Graphs in Unity.

3.2 Substance 3D Designer

Substance 3D Designer is an application that allows us to create 2D textures procedurally in a node-based interface. As a result, we can build textures without having to draw them ourselves, and we can easily change parameters of the whole texture without having to start over.

3.3 Houdini

Houdini is a 3D animation software used for the creation of complex visual effects like simulations, lighting and animations. It is most commonly used in filmmaking, but it can still find some usage in video game development. It also possesses a procedural, node-based generation system, like *Substance 3D Designer*. In our case, we will use it to create realistic fluid simulation.

4 BREAKING DOWN THE COMPONENTS

The type of explosion is the most important aspect of their implementation. Whether it is a fire detonation, an atomic cloud, a supernova or a magic burst, each will need different particle systems with different parameters. By adding stylization and performance into the mix, a lot of possibilities open up.

Despite everything, an explosion can be summarized in a few core components :

- the **blast** : deflagration of matter (fire, magic, poison, etc...);
- the **lights** : initial flash and other bright flares ;
- the **debris** : sparks, trails, flying rocks and dirt ;
- the **shockwave** : heat distortion, expanding ring ;
- the **remains** : lingering smoke, cracks and scorching marks.

Note that camera effects like screen shake, exposure or other post-processing effects are not counted as an explosion component due to the fact that they are more linked to the camera and not the VFX itself.

In this section, I am going to showcase each of these components and elaborate on the techniques used to implement them by creating and breaking down a realistic, fiery explosion.

4.1 Blast and Smoke

The blast is the most important component of the explosion, as it can define its own intensity, the type of the explosion and the element that is being expelled.

There are many ways to represent this effect. For example, a stylized explosion could incorporate the blast with the help of meshes of varying sizes, themselves having a special material that gradually dissolve and warp the mesh over time with the help of a texture (here, a simple Perlin noise). The meshes would be set to randomly rotate and be expelled outwards from their spawn point (See figure 3).

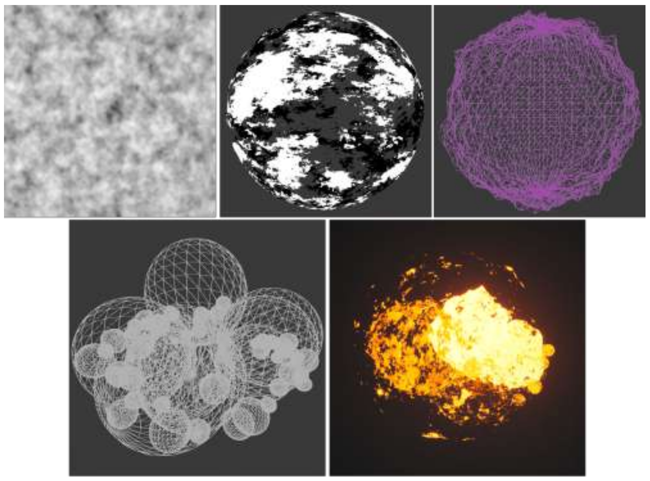
Figure 3: Spherical mesh with material using a Perlin noise as a dissolving texture, with the resulting blast.

In the case of our realistic explosion however, two elements go hand in hand into representing the blast : a burst of fire and the resulting smoke. To achieve the realistic flowing effect of both of these elements intertwined together, a fluid simulation is necessary.

A plugin called *Niagara Fluids* allows *Niagara* to make real-time fluid simulation. On its own, this simulation works great for a single explosion or for cutscenes. However, fluid simulations take up a lot of processing power and are not suitable if we want to spawn multiple VFX in real time. Giving the realistic style of the game it might be implemented in, other processing-heavy effects will definitely cluster up the memory further : frame rate is bound to be inconsistent, and the more of these VFXs that are spawned, the more the machine will have trouble calculating everything.

What we need to do then is to "bake" the simulation : pre-computing the simulation, so it can be exported as a simplified asset. The effect can no longer be modified unless being baked again, but it stays visually the same as the original simulation without having the need to be calculated each frame anymore. One of the most common ways to bake a visual effect is to capture the simulation frame by frame, effectively changing the 3D simulation into a 2D animation (doing so might sacrifice some of the effect's depth, but depending on the possible point of views it can be seen, this issue could be negligible). This sequence of frames is called a flip-book (See figure 4).

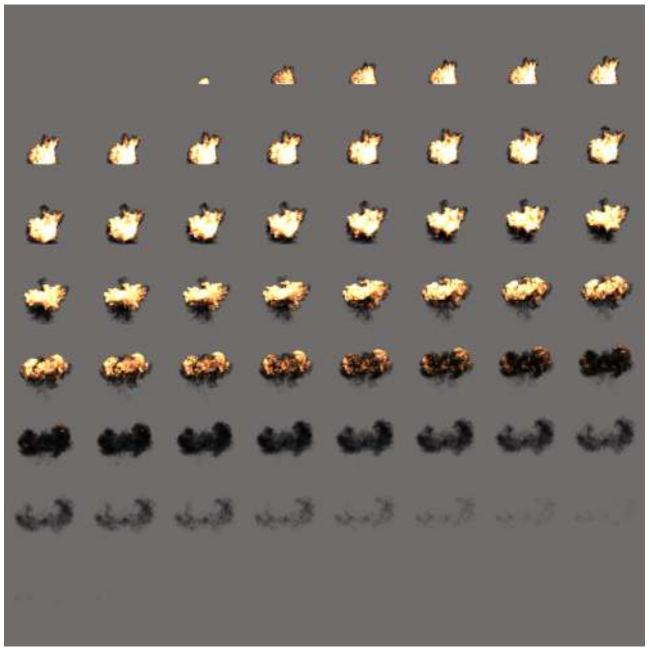

Figure 4: Flip-book of an explosion consisting of 64 different frames.

Niagara has the ability to bake effects on its own, transforming a fluid simulation into a flip-book. As efficient as it is able to simulate smoke, it however struggles to simulate a blast of fire. Moreover, it cannot bake the normals, making the effect lack depth. To resolve that issue, we instead simulate the smoke and blast with *Houdini*.

The *Houdini* software allows us to create and render a highly customizable explosion, from the power and velocity of the initial blast to the buoyancy and expansion of the smoke (See figure 5).

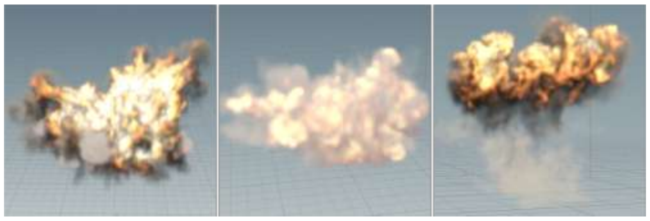

Figure 5: Three explosion simulations in *Houdini* with different parameters.

Once the parameters have properly been set, we can bake the simulation into a flip-book. A camera is set in the environment first, and the blast and smoke are both separated into different renderers. This allows us to have a flip-book for the smoke and for the blast separately, making it possible to be easily tweaked individually in *Niagara*. The normals of the smoke are also rendered into a flip-book of its own : they will map each frame of the flip-book by indicating the facing position of each pixel, allowing the smoke to be properly affected by light, giving it an accurate depth. The blast does not need its normals rendered, as it is already emitting its own light. We repeat this process with different simulation seeds to allow variety.

The flip-books of the fire and the smoke are both represented in grayscale, color not being an important parameter, especially if we want to manually input it inside *Niagara* without having to bake everything again. Being grayscaled, the Red, Green, and Blue RGB channels of the flip-book are all the same, and information is needlessly redundant. To optimize space and performance, we can then extract a single one of those channels and merge them together with the needed flip-book channels for a single explosion. Texture sampling is a costly operation, and merging all of our information into one texture will reduce the number of times it is used.

Using *Substance 3D Designer*, we import the needed flip-books and extract the channels, then merge them into a single texture.

In the case of our explosion, we use the red channel for the value of the smoke, the green channel for the smoke's opacity and the blue channel for the blast's value. In the end, we have two resulting flip-books for each explosion : the main values and the normals (See figure 6).

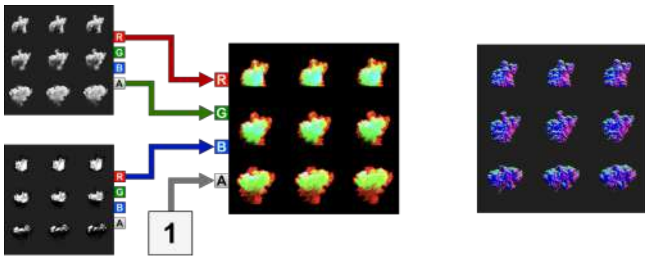

Figure 6: Diagram of the two final flip-books used when representing the blast and smoke. On the left is the smoke's value, opacity and blast value, respectively stored in the Red, Green and Blue RGB channel. The alpha is set to 1. On the right is the normals of the smoke.

In order to play any flip-books in *Niagara*, we need to create a material that stores the texture we need to use. In our case, since we stored our values in a customized way, we can use this step to unpack the texture's channels correctly into getting the final material. By using dynamic parameters, we can set the color of the smoke and blast separately. We also use custom color curves to set the color of the blast depending on its value over the frames ; this allows us to make the blast brighter at the peak of the explosion, and gradually make the color colder when it is dissipating.

Finally, we create instances out of this material in order to input all our different explosions, with their associated flip-books. The last step is now to create the particle system in *Niagara*. We spawn a single particle and set as much sprite renderers as we have different material instances. With the help of a custom variable, we set these renderers to only be visible depending on this variable's value, then randomly change the variable at each particle spawn. The result is a particle system that plays a random burst and smoke flip-book each time it is called.

At this point of the process, our explosion clips through the ground and other nearby meshes with a hard, flat intersection line, as well as hiding everything that is behind it. The desired result is for the VFX to have some gradual transparency depending on the surfaces next to it. As a final touch, we can then make our material use depth-fade in order to change our explosion into a soft particle.

A depth-fade makes objects semitransparent depending on their distance from other objects. This is achieved by adjusting their opacity based on how close they are to the surface behind them, using the depth buffer. After a depth-fade is applied, the explosion texture no longer clips through the ground with a hard edge, but rather with an opacity gradient. Similarly, objects can finally be blended inside the explosion and not be hidden by it, even when both are in the same position (Figure 7).

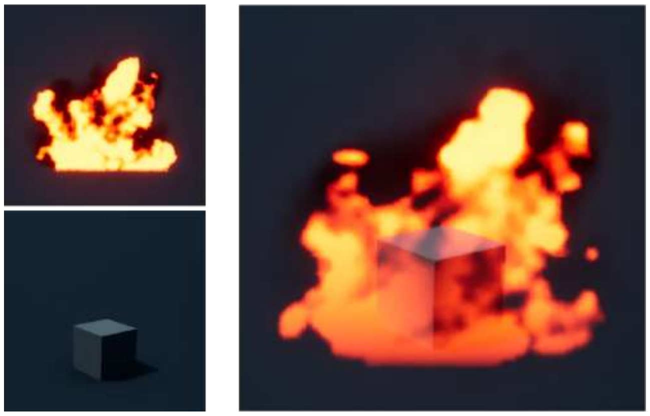
Figure 7: Effect of the depth-fade. On the left, the explosion hides the cube and clips through the floor with a flat intersection line. On the right, the cube blends correctly inside the explosion and does not clip through the ground anymore.

4.2 Lights

On its own, the blast and smoke particle system is not enough : many more elements need to be added to add more flair to the final VFX.

An initial flashing of light will help the explosion to be more sudden and gain intensity. Using *Substance 3D Designer*, I made a lens flare texture from scratch, then plugged it inside a material. Using custom color curves, we can adjust the color and opacity of the flare depending on its value. We then use the grayscale values to set the opacity and color of the material. Next, we use this texture in another *Niagara* particle system that will spawn a single particle. We set a very short lifetime for the particle to acts as a brief flash. By default, a sprite renderer always faces the camera. We then set the color of the material to follow a color curve in order to affect its value and transparency over its lifetime.

Colors can be represented with Hue, Saturation and Value (HSV) in *Unreal Engine*. It also possesses a volume : a component in the editor panel that can apply many different post-processing effects, one of them being bloom. Bloom can be easily influenced when tweaking the value of an HSV color - a color with a high value will naturally glow because of the bloom effect, effectively simulating light reaching the eye (Figure 8).

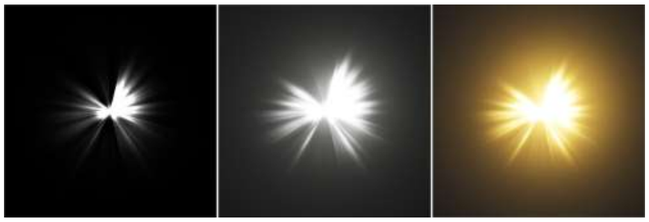
Figure 8: On the left, the initial texture ; in the middle, the adjusted texture with bloom ; on the right, the final flash texture with high color value.

The next step is to then simulate the light into the world itself. Conveniently, *Niagara* can spawn a light instead of sprites with light renderers. We can then create a separate particle system that will spawn a single light when the flash happens.

This solution is simple and effective, but it can cause some issues : lights can require lots of calculations, and depending on the amount of explosions we want to spawn at the same time, this can significantly hamper the machine's processing power.

Just like for the case of the fluid simulation, there are multiple solutions for this conundrum, depending on the context we want the explosion to be in :

- The complexity of the lighting brings an element of importance and focus to the explosion. If we want the explosion to be a notable event, the resulting lighting should be more accurate.
- If the explosion is always static, in a cutscene for example, the lighting can be baked beforehand.
- If the VFX is on a simple surface at a definite height and/or the light is small enough, the lighting can be simplified to only be a circle-shaped additive texture on the floor.

4.3 Scorching and Cracks

A sizable enough explosion almost always leaves a mark, and in the case of our fiery explosion, we are talking about scorching marks. We want the explosion to leave a specific texture along the surface under it that then gradually disappears over time. Doing the latter in *Niagara* can be achieved with a color curve over lifetime. The former, however, is more difficult to carry out : we need to use decals.

A decal is a material that is being projected onto meshes, effectively plastering a texture along a surface. *Unreal Engine* has a built-in decal system, allowing us to easily create them. However, what we want is to be able to render and influence those decals directly from *Niagara*, being able to fully treat them as particles.

We then need to make our own custom decal material, so it can be used inside *Niagara* onto a mesh particle. After creating the scorching mark and cracking ground grayscale textures in *Substance 3D Designer*, we then create a new particle system inside *Niagara* that will spawn a single static cube mesh. We adjust its original position to be lower than the origin of the explosion, in order to make the cube clip through the ground when spawned (See figure 9).

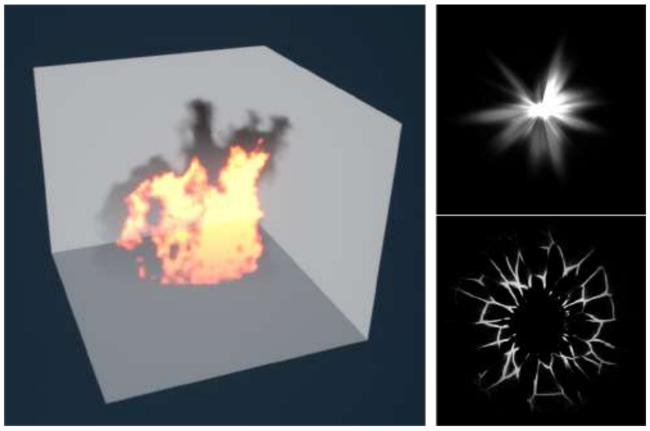
Figure 9: On the left, the particle cube mesh used to create the projection ; on the right, the two textures used for the scorching mark and cracking floor, respectively.

With our particle system ready, we can now start creating the shader for the material. We make the material transparent and two-sided, so it does not cull the mesh's back faces. We then disable the depth test for the material, so the decal does not disappear if the camera gets too close to the mesh.

The shader will rely on the scene's depth to project the texture's UV into the world. To achieve that, it will calculate the mesh particle position in local space ; however, instead of using the pixel position to render the texture on the mesh, it will calculate the position of the world surface behind the pixel with this equation :

$$\frac{S}{p}(W - c) + c$$

Where :

- W is the absolute World position ;
- S is the current Scene's depth ;
- p is the Pixel depth ;
- c is the Camera position.

From there, we can then scale and offset the UV in order to create a single tile of the texture. We set these offset and scale values as dynamic parameters, so they can directly be tinkered with *Niagara*. We can then freely rotate the mesh and input the transparency and color of the projection over time, like any other particle. Using color curves, we can also influence and adjust the color of both textures as well (See figure 10).

4.4 Debris

The debris of an explosion consist of the sparks, rocks, dirt, smoke trails and other flying objects that it generates. Depending on the type of surface the explosion happens on, it can be the source of different debris, like rising dirt, flying rocks or splinters for example. There are different ways to simulate those effects. For instance, *Houdini* can create and bake a 3D animation that can then

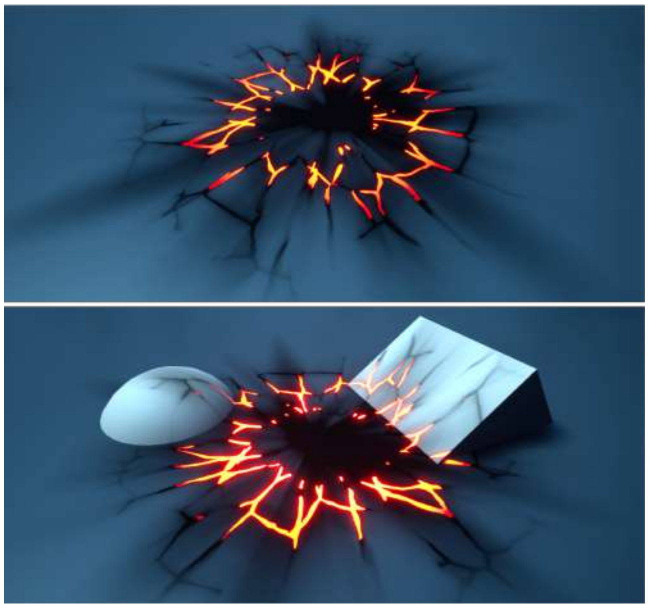
Figure 10: On top, the explosion's remains as decals used on a flat surface ; on the bottom, the same projection used on a more complex terrain.

be exported in *Unreal Engine*. Just as for the blast and smoke, we could also recreate these debris with flip-books again. While this can provide a solution, I wanted to explore alternative ways to implement them, without the use of *Houdini*.

Our realistic explosion here needs two more elements to increase its credibility : sparks and flying rock debris. Sparks can be easily implemented in *Niagara*, with a different particle system. What needs to be noted however is the efficiency of the particle system : it is made of many different particles, all simulated at once, and as such it can be heavy on our machine performance. To alleviate that issue we can simulate the particles not on our CPU (*Central Processing Unit*), but on our GPU (*Graphics Processing Unit*), as the GPU's parallel processing allows it to simulate millions of particles at once. Using the CPU over the GPU would allow for much more precise calculations, but with the amount of particles that we use and the simplicity of the effect, we do not need such accuracy.

For the rock debris, we can create our own custom rock mesh and follow the same logic as the spark particle system. However, we want to go the extra mile and make it so that the spawned particle meshes collide with the environment. This can easily be done inside *Niagara*, but there are multiple options on how collision is handled, depending on if the particle runs on the CPU or GPU. On the GPU, there are three options :

- Using analytical planes : one or two infinite planes are placed on an axis, independently of scene geometry. The particles will then only collide on it. This option is very efficient, but depends heavily on the context our explosion is used in. A static VFX happening on a flat surface is a perfect way to use this option, but if the terrain is more complex, this will lack accuracy ;

- Using the depth buffer generated by the current perspective. This option is very performant, but particles cannot collide with occluded or offscreen objects. It can also be unreliable if the camera moves too close or too quickly from the VFX. Collision might even get ignored if the particles move too fast ;
- Using the global distance fields. Distance fields are a very low-cost simplification of the scene's data around the other scene meshes, mainly used for calculating more efficient shadows. The resulting mesh is rough (See figure 11), and the accuracy gets reduced the farther away from the camera the objects are. As a result, collision might be inaccurate on far away objects, or not even collide at all. Unlike the depth buffer option, however, collisions can happen offscreen.

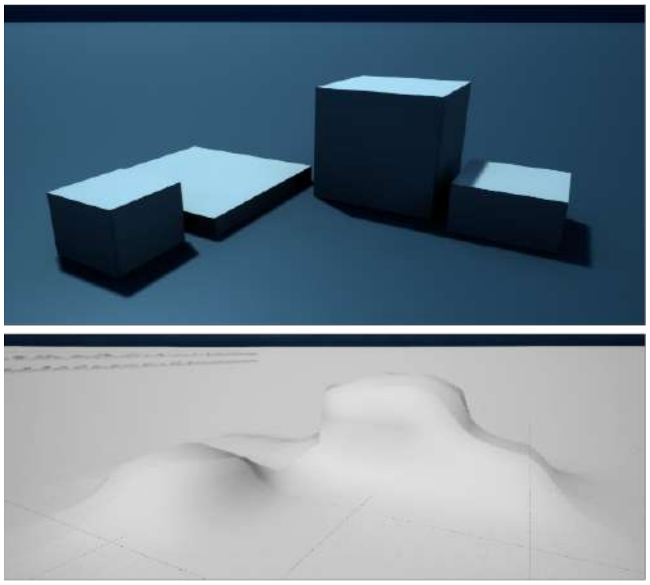
Figure 11: A terrain with its corresponding visualization in global distance fields on the bottom.

On CPU-handled particles, there are only two options for managing their collisions : analytical planes and ray tracing.

The analytical planes option is similar to the GPU one. Ray traced collisions are more expensive on the processor, but more accurate. Being handled by the CPU, particles can selectively collide with specific objects in the scene.

To summarize, CPU particles are expensive but accurate, and GPU particles are more efficient but less accurate. In the case of our explosion, the rock debris uses the GPU-handled global distance fields as reference for managing collisions. This will keep the effect cheap on memory while being more accurate than the depth buffer option, and still being able to be used on complex terrain (Figure 12).

4.5 Shockwave

The final component of an explosion, the shockwave, can add a lot of punch to the final VFX. A common answer to a stylized shockwave would be to spawn a disk mesh particle at the center of the explosion, with a custom texture that clamps to the disk borders. The disk will then simply grow over time and dissipate at the end of its lifetime (See figure 13).

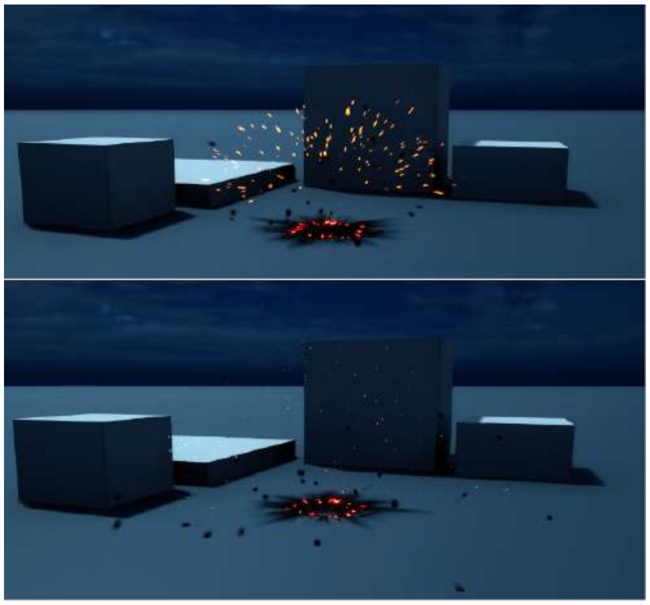
Figure 12: The final sparks and debris of the explosion. The debris are colliding with the floor, resting along its surface.

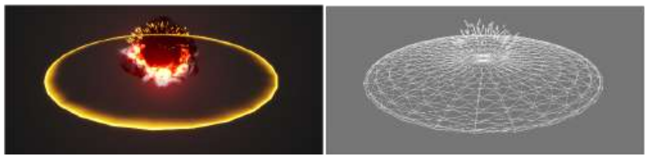
Figure 13: A shockwave with its corresponding wireframe. The shockwave is but a flattened sphere with a texture going along its border.

This works well for stylized explosions, but it might be too much for our realistic explosion. An important aspect of the shockwave that can be used in both stylized and realistic explosions, is the heat distortion made by said shockwave.

Following the same logic as a stylized shockwave, we need a rapidly growing sprite particle that distorts the scene behind the explosion. First, we make the needed texture in *Substance 3D Designer*, then create a custom material in *Unreal Engine* to create our warping effect.

The material will be fully transparent, but its normals will differ from a standard flat surface. On its own, this has no effect. However, when coupled with the material refraction, the environment will effectively be wrapped behind the material, as it bends light going through it. We can then use the shockwave texture as a mask in order to apply this warping effect only on the texture. In the end, we have a functioning heat distortion shockwave for our explosion (Figure 14).

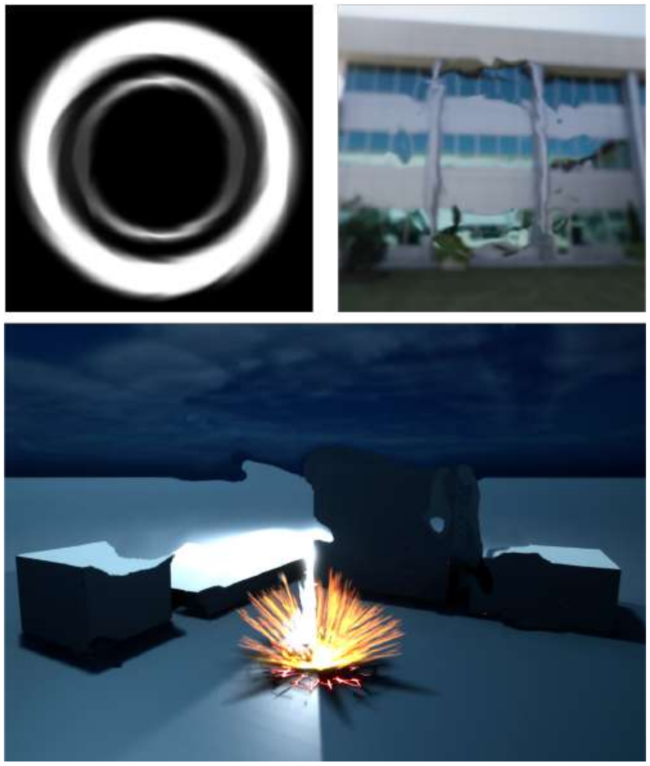
Figure 14: The shockwave texture, an example of the ring heat distortion and the final shockwave used in real time.

5 CONCLUSION

After adding some additional smoke particles and polishing the parameters of each particle system, our explosion VFX is complete (See figure 15).

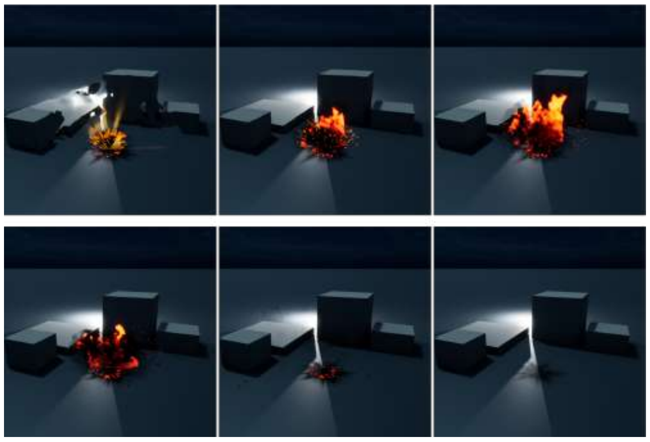
Figure 15: Final result for the realistic explosion VFX.

As stated before, VFXs (and by extension, the way they are implemented) can change a lot depending on the context they are used in, but also the type of effect desired. A realistic fiery explosion will be implemented differently than a stylized atomic bomb explosion, a magical explosion or a supernova for example. All these effects might follow the same structure, and some building techniques can be recycled as a result ; but they each hold specific traits and details that will make their implementation unique from one another.

The approaches discussed in this report are merely a few of the plethora of other parameters and methods available in VFX making. A lot of time was spent researching, from learning how to use the tools to trying out new tricks in the art of VFX. And all of that was just scratching the surface of what is available. Unfortunately, due to a lack of time, I was not able to explore different types of explosions. Despite that, this project taught me a lot about VFXs and made me eager to learn even more about all the ways they are made, and how we can make them stand out. I hope this research fueled your curiosity as much as it did mine.

REFERENCES

[1] Adobe Substance 3D. 2021. *First Steps: Substance 3D Designer.* Adobe. https://www.youtube.com/playlist?list=PLB0wXHrWAmCzPPogfeSk3lhOsmflTfRv5
[2] aboutvfx. 2017. *Unity stylized nuclear explosion.* Real-Time VFX. https://realtimevfx.com/t/unity-stylized-nuclear-explosion/3626
[3] Adobe. 2023. *Substance Designer 3D Official documentation.* Adobe. https://helpx.adobe.com/substance-3d-designer/home.html
[4] VFX Apprentice. 2018. *Artistic Principles of VFX.* VFX Apprentice. https://www.youtube.com/playlist?list=PLQD_sA-R5qVKVYw3EVuRT7fSJsVukLEhD
[5] Niels Dewitte. 2020. *[Niagara 4.25] Particle Decals Mini Tutorial.* Real-Time VFX. https://realtimevfx.com/t/niagara-4-25-particle-decals-mini-tutorial/13177
[6] Unreal Engine. 2023. *Niagara Official documentation.* Epic Games. https://docs.unrealengine.com/5.2/en-US/creating-visual-effects-in-niagara-for-unreal-engine/
[7] Unreal Engine. 2023. *Niagara Tutorials.* Epic Games. https://docs.unrealengine.com/5.0/en-US/tutorials-for-niagara-effects-in-unreal-engine/
[8] Unreal Engine. 2023. *Unreal Engine Official website.* Epic Games. https://www.unrealengine.com/en-US/
[9] Pub Games. 2014. *UE4 - Making Fire - Screen Space GPU Particles.* Pub Games. https://www.youtube.com/watch?v=hZLbGvtyS6g
[10] Andreas Glad. 2016. *Realtime VFX Dictionary Project.* Real-Time VFX. https://realtimevfx.com/t/realtime-vfx-dictionary-project/570
[11] Houdini. 2020. *Destruction FX | 3 | Pyro FX | Burst Source and Solver.* Houdini. https://www.youtube.com/watch?v=-OJi-9YOY9M
[12] Houdini. 2023. *Houdini Official website.* Houdini. https://www.sidefx.com/
[13] Simon Houdini. 2021. *Pre-Baked Destruction in Unreal Engine || Houdini Tutorial.* Simon Houdini. https://www.youtube.com/watch?v=exreMc7CNiU
[14] Jason Keyser and Patty Ruhnke. 2023. *An introduction to choosing colors in game VFX.* Academy. https://www.gamesindustry.biz/an-introduction-to-choosing-colors-in-game-vfx
[15] Epic Online Learning. 2022. *Niagara Collisons.* Epic Online Learning. https://forums.unrealengine.com/t/niagara-collisons/518557
[16] Yiming Liu. 2020. *VFX UE4 Explosion Effect.* Real-Time VFX. https://realtimevfx.com/t/vfx-ue4-explosion-effect/12054
[17] Pawel Margacz. 2021. *How to Realtime - VFX / Explosion Breakdown / Unreal Engine 4 / Houdini.* 1MaFX. https://www.youtube.com/watch?v=YGBBsWbi8KM
[18] League of Legends VFX Discipline. 2017. *The complete guide to creating visual effects within League of Legends.* Riot Games. https://nexus.leagueoflegends.com/wp-content/uploads/2017/10/VFX_Styleguide_final_public_hidpjqwx7lqyx0pjj3ss.pdf
[19] Simon Schreibt. 2015. *Fallout 4 – The Mushroom Case.* Simonschreibt. https://simonschreibt.de/gat/fallout-4-the-mushroom-case/
[20] Gregory Silva. 2018. *Making Zelda-like VFX with Unity.* 80 Level. https://80.lv/articles/making-zelda-like-vfx-with-unity/
[21] Escape Studios. 2021. *Advanced VFX Simulations | Houdini Tutorial.* Escape Studio. https://www.youtube.com/watch?v=joOLqxu20QU
[22] Houdini Tutorials. 2016. *pop sop dop? what do these mean?* Houdini. https://www.sidefx.com/forum/topic/43515/?page=1
[23] FX Cat UA. 2021. *Explosions All.* Epic Games. https://www.unrealengine.com/marketplace/en-US/product/allexplosions
[24] Enrique Ventura. 2023. *Unreal 5.1 - Creating cinematic VFX from scratch. Chapter 3: Explosions!* Enrique Ventura. https://www.youtube.com/watch?v=LmXOLASrb4M

[25] Aleksey Zaitsev. 2020. *Another stylized explosion*. Real-Time VFX. https://realtimevfx.com/t/another-stylized-explosion/12507

ADDITIONAL RESOURCES

The entire project's files can be found on GitHub : https://github.com/NathBroch/Explosion-VFX