VFX Demonstration: Close Encounter of the Second Kind

Anthony Medina[*]
amedin12@ucsc.edu
University of California, Santa Cruz
Santa Cruz, CA, U.S.A.

Jan Yu[†]
jyu92@ucsc.edu
University of California, Santa Cruz
Santa Cruz, CA, U.S.A.

Malcolm Riley[‡]
masriley@ucsc.edu
University of California, Santa Cruz
Santa Cruz, CA, U.S.A.

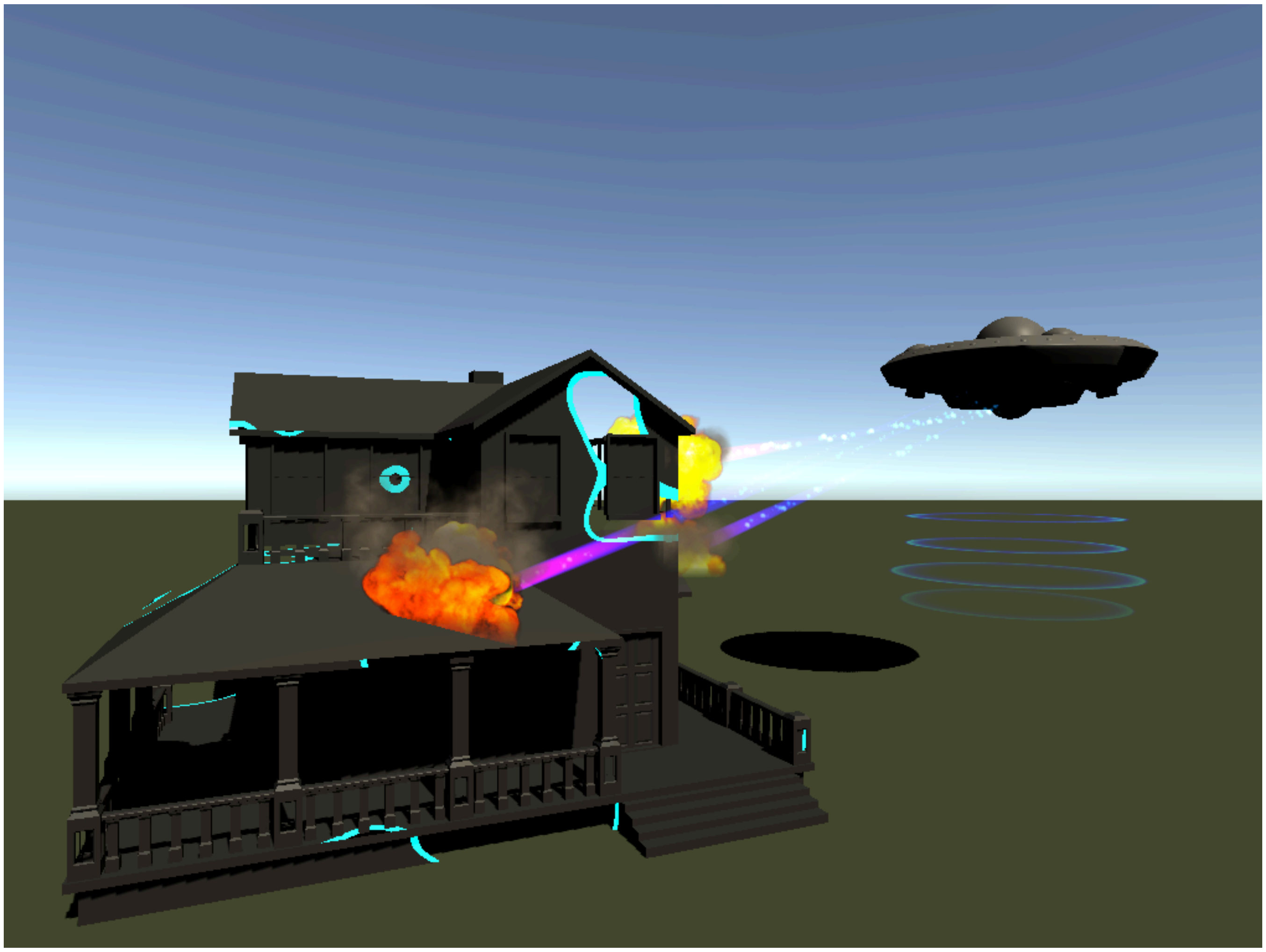

Figure 1: Scene View

ABSTRACT

Our project is a small vignette featuring a UFO firing an "energy weapon" at a farmhouse. As the bolts of energy impact the farmhouse, explosions abound and the house gradually dissolves.

[*]Explosion Particle Effects
[†]House Dissolve Effect
[‡]3D Models, Energy Weapon VFX, Hover VFX, Main Scene Assembly, Documentation

© 2019 Association for Computing Machinery.
ACM ISBN ?????????...$0.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

CCS CONCEPTS

• **Computing methodologies → Modeling and simulation**; • **Human-centered computing → Visualization**; Human computer interaction (HCI).

KEYWORDS

keywords,keywords,keywords

ACM Reference Format:
Anthony Medina, Jan Yu, and Malcolm Riley. N/A. VFX Demonstration: Close Encounter of the Second Kind. In *UC Santa Cruz '19: CMPM 163 Final Project, June 01–13, 2019, Santa Cruz, CA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 EXPLOSION VFX

Written by Anthony Medina. To get the exlosion effect, one must make use of Unity's particle systems. There are essentially two particle systems at work here, one for the explosion and an additional for the smoke effect. For the explosion and smoke, we would need to have some material that replicates the explosion effect and smoke. For this, I made use of the Unity particle pack and used a textured sheet animation from a fire explosion material and dust material. Starting with a basic particle system, we can check the renderer and add the material there. There will be a module that let's us edit the textured animation sheets and from there we can turn it on and set it up so that it will show the correct images, both the fire explosion and smoke being 6x6 tiles, which we set it to. Otherwise, we'll get the entire textured sheet to be displayed instead of each individual image. With this set, the effects are ready to be modified and altered to achieve the effect we're looking for. For the explosion, we should set the shape off and keep it natrually occurring. Next, we would want to keep it set to just one explosion so we would set the maximum amount of particles to 1 in the main settings of the particle. Here, we can also zero the start speed too in order to keep it stable and centered. In the emmission module, we would want to zero the rate over time and add a burst effect, keeping the minimum and maximum at 1. After thatwe can edit the simulation speed and duration of the explosion to your liking. Additionally, the color over lifetime can be adjusted depending on the effect you're dealing with, but just know that the module lets you literally change the color over the lifetime of your effect. Also, if you were to use the shape module, changing the shape to sphere would be the way to go since it gives the most realistic effect of an explosion than any other shapes. Changing the velocity over lifetime affects the speed of your particle over time and dampens it an arbitrary value. Finally, the size over lifetime module can be altered to just make the effect smaller or bigger over its lifetime depending on the effect you want. I wanted a small controlled explosion so I ended up removing and altering features here and then until I got the effect I wanted. For the smoke, we would have a similar approach. First, we want to adjust the angle of the particles so they move upwards an do this in their transform settings, and also make the maximum amount of particles something reasonable like 50. Then, we would change the emmission module to be 0 for the rate over time and once again set the burst option, minimum and maximum set at 50 or whichever max particles you have set. To give randomness for the smoke, we will then add a random between two constants in the start speed and edit there as needed. I set mine to 0 and 1 which will tell Unity to choose a random value between 0 and 1. After that, we can do the same to the start lifetime and I set my values to 2 and 3. Once again, to achieive an even more natural feel to the smoke, we can set the start rotation to a random value from -180 to 180 using the same option as before. Next, we can edit the shape to be either cone or spehere. I set mine to cone and made the base small with a wide angle. Either way, we're setting the smoke to appear as though it's coming out of the explosion so be sure to adjust the position with the explosion's. Now we should modify the color over lifetime module so that we can get a more realistic look. I set my color to start off dark and get a little lighter over time, having a dark gray turn into just gray. Lastly, I turned on force over lifetime and set the z axis to be random between two constants from .5 to 1 so that the smoke rises. With all of this combined, there should be a repeating explosion with some smoke surrounding it. Last edits I made were to cut looping from both effects to just get the one explosion effect I want, and editing the duration of the explosion and smoke. I let the smoke have a longer duration than the explosion so that their times were more in sync. Depending on the look you want of the effect, you can change the texture sheet animation or edit the render options, as well as modifying the particle itself. Last note would be to make sure that cast shadows recieve shadows is off always since we are really creating light and don't want that effect. Overall, explosions are a burst of particles that fade away over time while changing colors, from a bright light, to dark red, and ending off with a black smoke.

2 HOUSE DISSOLVE VFX

Written by Jan Yu. When I was making the dissolve effect, I tried to do something different by using a new feature in Unity known as the shader graph. This allowed me to better organize and see my shader at a higher level. In order to use the shader graph feature, I needed to import both the Shader Graph and the Lightweight Render Pipeline packages in Unity. I would then edit the render pipeline and upgrade the project materials to LightweightRP materials. Lastly I would create a new material based on the lightweight render pipeline to replace the standard materials. Using the shader graph, I created a simple noise to simulate the effects I want for dissolving an object. To create edges on the noise, I also added a step node that also takes from the simple noise and added a color to it. On the master node in the shader graph, I assigned the noise as the object's Alpha and assigned the edge effect created by the step node as the object's Emission. When I initially created the shader effect, it was made to dissolve in and out repeatedly by remapping the vector using the time node. But what I wanted in the project was to make it so that the house dissolves upon interaction and only once. In order to do this, I needed to create a script to control the properties of the shader. In the shader graph, there is a blackboard that allows me to create vector nodes with exposed parameters. I then used it as an extension to some of the nodes in the graph for me to control through the script. I replaced the sine time that I was using to loop the noise effect in and out in exchange for a vector with an exposed parameter that I can control.

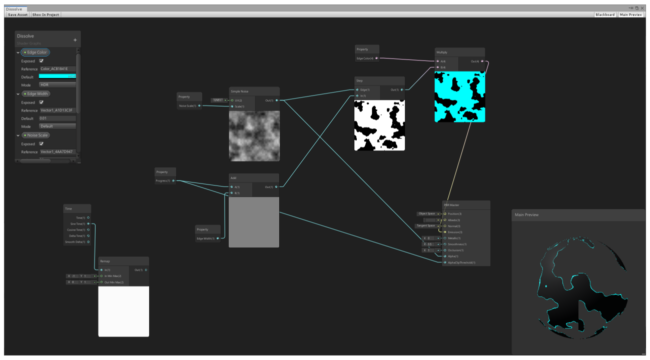

Figure 2: Unity Shader Graph.

To blackboard to the top left is what allows me to expose the parameters. Within the script, I made it so that upon interaction or mouse click, the house would dissolve only once. To do this, I used a smooth delta time to control the speed in which the house dissolves and since it doesn't bounce back, it would not loop back to its original form. The use of time is very important when dealing with noise because the way the Alpha Clip Threshold reads noise is that it takes a value between 0 to 1 and decides whether the object is transparent or not. By using the time to transition from 0 to 1, we can create an effect that gradually makes parts of the object transparent. The end result is a house desolving upon interaction.

3 UFO VFX

Written by Malcolm Riley. The primary focus of my visual-effects efforts was in the implementation of the UFO's energy weapon effects. There are four components to this effect: A modified rim shader applied to an oblong mesh forms the primary mass of the projectile, a considerably simpler transparent shader applied to a hemispherical mesh form the shockwave effect, and two simple particle systems form the trail and sparkles, respectively. The implementation of the latter two effects were trivial; being performed using Unity's built-in unlit particle shaders they will not be discussed in depth. Likewise, various other visual effects tasks were performed in the creation of this scene and will be discussed briefly in the "Miscellaneous Effects" section.

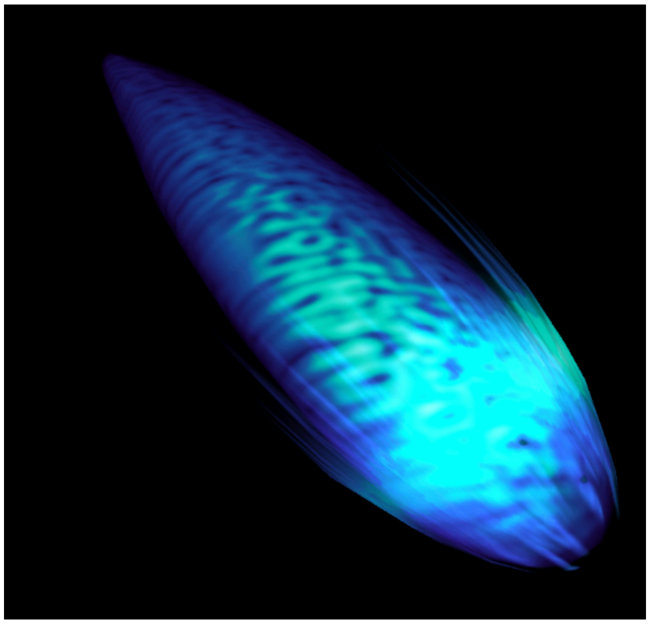

Figure 3: The primary energy projectile effect.

3.1 Modified Rim Shader

To add visual interest to the main mass of the energy projectile, a modified rim shader was used. As with any rim shader, the primary driving operation is the dot product of the normalized surface normal with the normalized view vector. However, it became desirable to modify this result by adding a screen-space noise sample and an intensity factor. As such, both the "rim incidence" vector and the noise sample were raised to arbitrary powers and multiplied.

The noise-enhanced rim incidence would thus given by the expression:

$$R = (\hat{\mathrm{I}}_{\mathrm{N}} \cdot \hat{\mathrm{I}}_{\mathrm{V}})^k \times T^j$$

Figure 4: Rim incidence vector equation.

Where $\hat{\mathrm{I}}_{\mathrm{N}}$ is the normalized surface normal, $\hat{\mathrm{I}}_{\mathrm{V}}$ is the normalized view vector, k is an arbitrary "Rim Intensity" factor, T is the noise texture sample color, and j is another arbitrary intensity factor.

In CG, this calculation is performed as follows:

```
pow(dot(input.normal, input.view), _RimIntensity)
* pow(noise, 1 - _NoiseIntensity)
```

Figure 5: CG code for the rim incidence vector calculation. The expression used to calculate the noise-enhanced rim incidence vector, in CG code.

The components of this noise-enhanced rim incidence vector were then averaged and then used as the UV parameter for a gradient texture lookup, allowing for an arbitrary color gradient across the surface of the mesh.

Under most circumstances this calculation would be sufficient, but due to the multi-color nature of the gradients used, there were undesirable visual artifacts around the very edge of the mesh – where the surface normals were perpendicular, or very nearly perpendicular, to the view normals. It therefore became necessary to clamp the components of the rim incidence vector to a fractional nonzero value.

As a finishing touch, the alpha of the returned color was also related to the noise-enhanced rim incidence vector, creating greater opacity towards the center of mass of the mesh, and lesser opacity towards the visible edges.

3.2 Simple Transparent Shader

A considerably simpler effect was used to create the shockwave effect that precedes the projectile. This was created with a one-sided hemispherical mesh, deliberately UV-unwrapped by a means that is commonly considered "wrong".

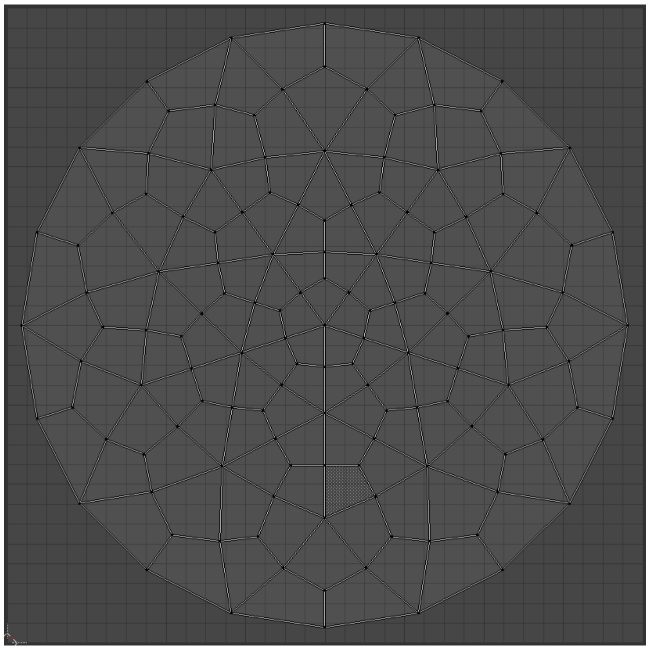

Figure 6: "Incorrect" UV Mapping of hemisphere mesh.

While it is typical to add seams to various edges of meshes in order to minimize texture stretching across a mesh's surface, it was in this case deliberately avoided. After some experimentation with transforming texture UVs to polar coordinates it was determined that this approach would be simpler (and likely computationally less expensive for the GPU) than trying to map spherical polar coordinates to a more ordinary wrapped rectangular texture. As such the texture was prepared to similarly be distorted in a circular pattern towards the center.

To complete the effect, the texture values from the original sampled texture were mapped to a color gradient texture for the final fragment output, by the same method as in the previous section.

On retrospection, it is entirely possible that this is, in fact, the standard approach to such a problem, but for me it was an important lesson: sometimes the simple, "less-correct" approach is the correct one.

3.3 Miscellaneous Effects

In the creation and assembly of the final scene, a number of "more trivial" visual effects were implemented. They are discussed below in no particular order:

- **Weapon Trails:** This was accomplished by use of two particle systems. The first particle system was configured to only render the particle trail, with some modifications made to alter the color and alpha of the emitted particles over the length of the trail. The second particle system was a simple world-space spherical emitter with color-over-lifetime and size-over-lifetime parameters configured. In both cases the particles used Unity-provided standard unlit shaders as they were sufficient for these trivial effects.
- **UFO Hover Effects:** This was accomplished by a simple script and particle system. A script sets the object transform position to some sine-modulated offset using time as an input parameter. Though it would have been possible to implement this effect as a vertex displacement shader, this was undesirable as the actual position of the object was needed for other effects, such as firing the weapons (to determine the origin and rotation of the spawned energy weapon object) or emitting the hover particles. The hover particles are, again, a simple world-space conical emitter with color-over-lifetime and size-over-lifetime parameters configured, using the Unity-provided standard unlit shader.
- **Weapons Fire:** To perform the weapons-fire-from-saucer effect, a distance-capped raycast was performed from the camera into the scene on click, and an impact point of this raycast was determined. From this impact point and the (known) world-space transform of the flying saucer's undercarriage, a velocity vector was calculated between the two points and then applied to the newly-spawned energy weapon projectile.

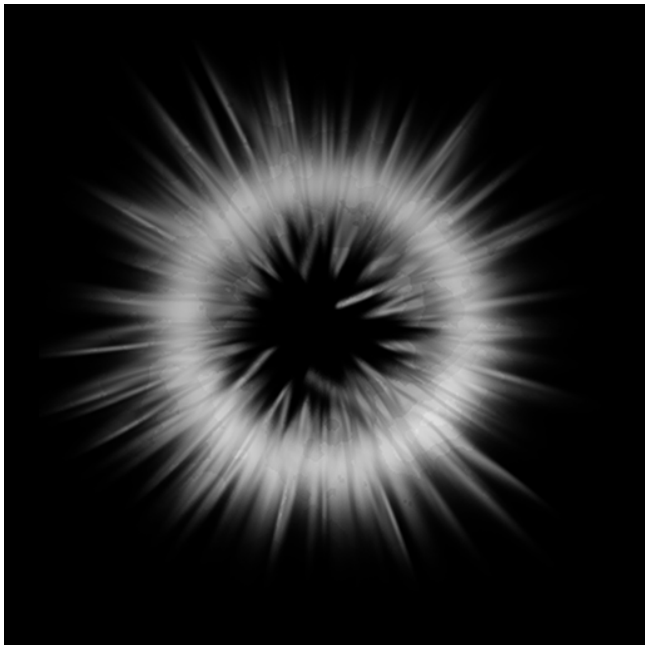

Figure 7: Texture applied using "Incorrect" UV Mapping

- **Weapons Impact:** This was a matter of attaching physics components to all of the relevant objects, and implementing a script to detect collision: The house mesh received a mesh collider, and each energy weapon projectile received a capsule collider. When collisions were detected between these families of entities, a script incremented the dissolve parameter of Jan's shader, and spawned an instance of Anthony's explosion, as well as destroying the actual projectile instance. The particle emitters for the projectile instances were permitted to linger momentarily in order to preserve the projectile trails.