Atmospheric Entry – Geometry Shader

BRIAN VAN HYFTE

Research

Introduction

When an object enters the atmosphere, it heats up due to friction with particles. This effect is called atmospheric drag or aerodynamic heating. Because of the velocity needed to generate the effect, there is no footage of it up close, making it harder to recreate. What we do know, is that it creates a trail of fire behind it.

We will be trying to recreate this. Most notably the surface heating up, and the trail of fire behind it.

This effect has been recreated in the game Kerbal Space Program, which is what I will try to reproduce.

Effect

The effect for the shader can be broken down into 4 major pieces. The object heating up, creating an orange-red hue on the surface of the object, a trail of fire bouncing off the front of the object and following it, the smoke followed by the trail of fire, and small chunks of debris flying off the object.

The smoke and debris are easily done in a particle system, and thus will not be included in this shader. We will be focussing on the fire trail, and the surface heating.

Surface heating

This is a simple effect we can achieve with a pixel shader. We'll have to ensure that surfaces directly exposed to friction are hotter (more white) than surfaces with an angle (more red). Density of the atmosphere could also play a role in how fast a surface heats up. There are 2 ways I could establish this effect. I can use the dot product of the surface, to check if the surface is in line with the airstream. Based on the angle, I can heat a surface up. I would have to use shadow mapping to ensure it only happens to directly exposed surfaces however, and when testing this, there would be issues with hard edges, as some surfaces had sharp angles. Another way easier technique is to simply place an orange light along the velocity vector.

Trail

This will be the main bulk of the shader. We'll have to use a geometry shader to extrude planes along the airstream, whilst ensuring it is only happening to surfaces directly exposed to the airstream that are on the edge of the object. We could achieve this with something similar to god rays. We could use a silhouette shader to determine the edge of the model. The developers of KSP generated a shadow map along the airstream to determine if a plane is exposed to it.

Silhouette shader

Using a silhouette shader, we could determine the edge of the model, and use this information to extrude planes. However, when researching, I found that most silhouette shaders don't only show the edge of the model from a certain perspective, but rather highlight all the edges of the model. This is not what we want, we only want the outline of the model from the airstream's "perspective". There are probably ways to achieve this, but the shadow map way seemed to be more straight forward, as I already had implementations for this in the engine.

Shadow mapping

When using shadow mapping, instead of generating a shadow map from the perspective of the light, we use the perspective of the airstream. We can use this information to check if an area is directly exposed. Combine this with a dot product, and we have a way to know what planes to extrude.

Implementation

Pass 1 – Surface heating

We start off with the pixel shader. For the sake of this demo, I will implement specular, diffuse and ambient lighting. You can easily expand the shader to support normal mapping, environment mapping, etc. However, for this demo, those are not necessary.

For specular, we just use the phong algoritm.

```
float3 CalculateDiffuse(float3 normal, float2 texCoord, float3 lightDirection)
{
    float3 diffuseColor = gColorDiffuse;

    //Diffuse Logic
    float diffuseStrength = saturate(dot(normal, -lightDirection));
    diffuseColor *= diffuseStrength;

    if (gUseTextureDiffuse) {
        diffuseColor = gTextureDiffuse.Sample(gTextureSampler, texCoord) * diffuseStrength;
    }
    return diffuseColor;
}

float3 CalculateSpecularPhong(float3 viewDirection, float3 normal, float2 texCoord, float3 lightDirection)
{
    float3 specularColor = gColorSpecular;
    if (gUseTextureSpecularIntensity) {
        specularColor = gTextureSpecularIntensity.Sample(gTextureSampler, texCoord);
    }
    //Specular Specular Logic
    float3 reflectedVector = reflect(lightDirection, normal);
    float specularStrength = dot(-viewDirection, reflectedVector);
    specularStrength = saturate(specularStrength);
    specularStrength = pow(specularStrength, gShininess);
    specularColor *= specularStrength;

    return specularColor;
}
```

We calculate the entry heat by simply adding these 2 together. However, we run into an issue: planes not directly exposed can still light up. To fix this, we use the generated shadow map from the airstream's perspective. To calculate the shadow, we use the following function.

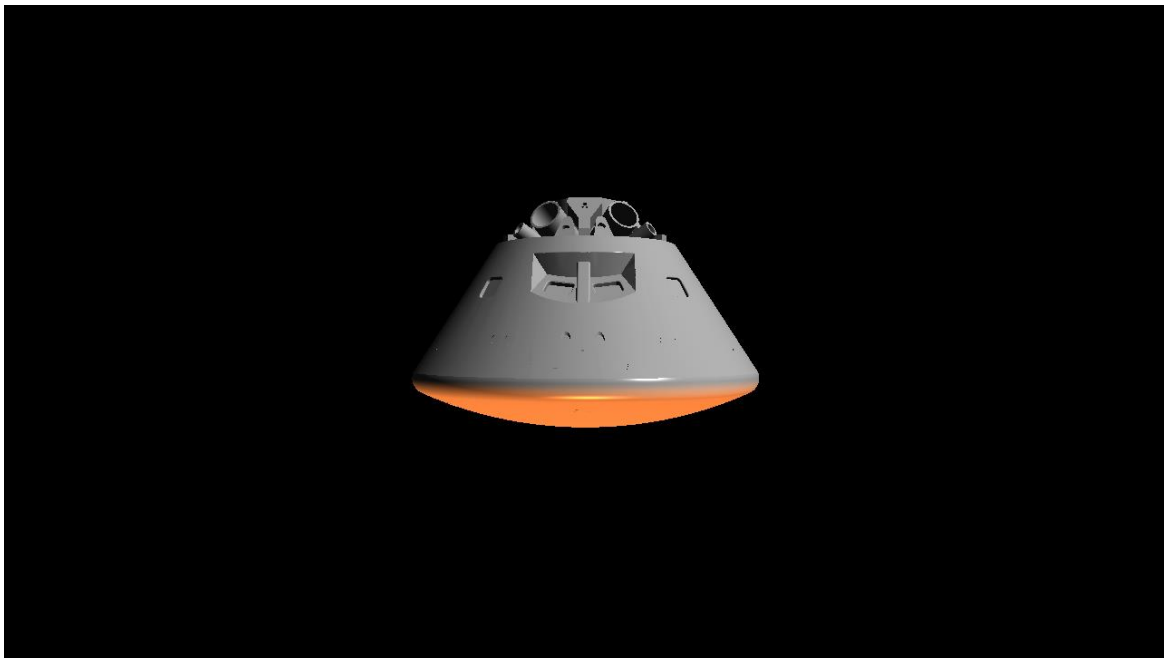
```
float CalculateShadow(float4 lightPos, Texture2D shadowMap, float bias)
{
    float4 lpos = float4(lightPos.xyz / lightPos.w, lightPos.w);
    if (lpos.x < -1.0f || lpos.x > 1.0f || lpos.y < -1.0f || lpos.y > 1.0f || lpos.z < 0.0f || lpos.z > 1.0f)
        return 1;

    lpos.x = lpos.x / 2 + 0.5;
    lpos.y = lpos.y / -2 + 0.5;
    lpos.z -= bias;
    float sum;
    float x, y;
    for (y = -2; y <= 2; y += 1.0f)
    {
        for (x = -2; x <= 2; x += 1.0f)
        {
            sum += shadowMap.SampleCmpLevelZero(cmpSampler, lpos.xy + texOffset(x, y), lpos.z);
        }
    }
    float shadowFac = sum / 25.0f;
    return saturate(shadowFac + 1 - gShadowStrength);
}
```

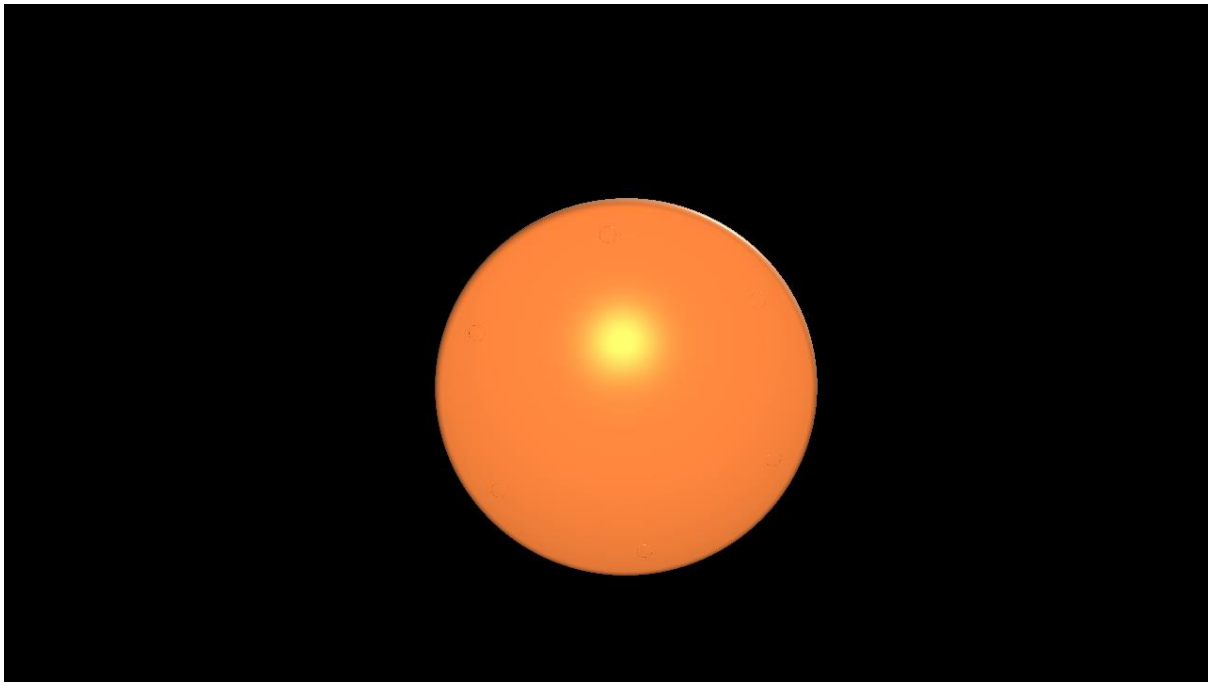
This function will return 1 if a surface is completely exposed, and a 0 if it isn't. Depending on angle, it can be between those values as well. Knowing this, we can use this value as a multiplier on our entry heat, as any surface directly exposed will just multiply by 1, but any surface not exposed will just multiply by 0, thus putting entry heat at 0. This is not the only thing we want to multiply our entry heat with though, we want some variables to work with. First off, we use an intensity variable set by the user. This can be used if an atmosphere is denser. Secondly, we want to multiply it with the velocity, as objects with higher velocities will accumulate more heat.

This is the resulting code, combined with a global light and the ambient lighting.

```
float3 entryHeat = CalculateDiffuse(normal, input.texCoord, gVelocityDirection)
    + CalculateSpecularPhong(viewDirection, normal, input.texCoord, gVelocityDirection);
entryHeat *= gEntryLightColor.rgb*gEntryLightColor.a;
entryHeat *= velShadow; //Add shadows so it only shows light on planes that are directly exposed to the velocity vector
entryHeat *= gEntryIntensity; //Strength of the light. Atmospheric density can change friction, which means surfaces heat up more
entryHeat *= gVelocity; //A faster object has more friction, thus it is hotter

return float4(ambient + globalLight + entryHeat, 1);
```

And here you can see the effect in the engine.

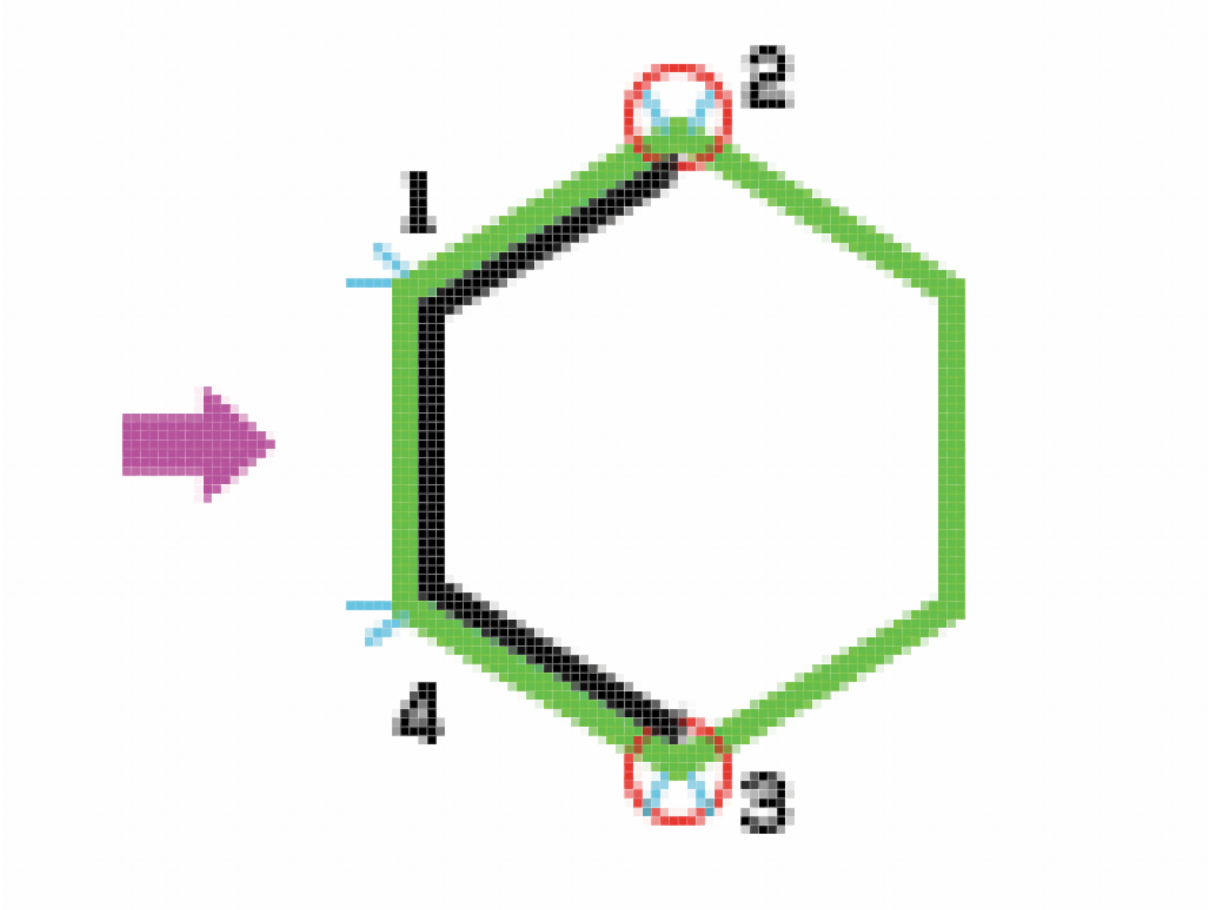




We can play around with the intensity to have a bigger heat effect, but this will do.

Pass 2 – Fire trail

This will be the main bulk of the shader, and the geometry shader. The biggest challenge is to ensure the trail only gets extruded on edges. To do this, we will use the shadow map we generated earlier to see if a vertex is directly exposed to the airstream again. This time, we will also have to consider the surface's normal and the airstream vector. There should only be an effect if the vertex is occluded, but the angle also must be lower than, or equal to 90°. Here's a diagram explaining it



Let’s dissect the image:

- Purple arrow: Velocity vector
- Green hexagon: Object entering atmosphere
- Black edge: Occluded on shadow map
- Blue stripes: Normal on vertex (will probably be average on actual model)
- Red circle: Vertices that get extrudes

So, what exactly happens in this situation? The front 2 points (p1 and p4) both meet the occlusion requirements. However, angle between the velocity vector and their normals is larger than 90°**.**Thus these vertices do not get extruded. If we look at the outer points (p2 and p3), these points both also meet the occlusion requirement. If we average the normals, it will form a 90° angle with the velocity vector, thus these vertices will be extruded. The other points are not occluded, so they will not be extruded either.

This is the basic principle. Let’s look at the implementation.

We will treat our shader on an edge basis, instead of a vertex-by-vertex basis. This way we can easily extrude an edge, and we won’t run into issues with vertex density.

This trail must show a colour change, from bright (Almost white) orange to dark red. To achieve this gradient, we generate 2 quads stacked on top of each other. We start off by creating vertices on both points of the edge, giving them the bright colour, as the fire will be hottest at this point. From here, we extrude a quad along the velocity vector with a colour that is lerped between the bright colour, and the dark red colour. We multiply the velocity vector with a velocity multiplier, so the user can easily change the length of the trail. We also multiply it with the dot product, so edges with lower angles will generate a longer trail. Lastly, we multiply it with the actual velocity, and the occlusion.

This will give us an orange trail, with a red end. However, we want to be able to change the length of the red, so we add another quad that just uses the red colour. We multiply the previous vertex position with a modifier, so the user can set the length of the dark red.

This is what that code looks like.

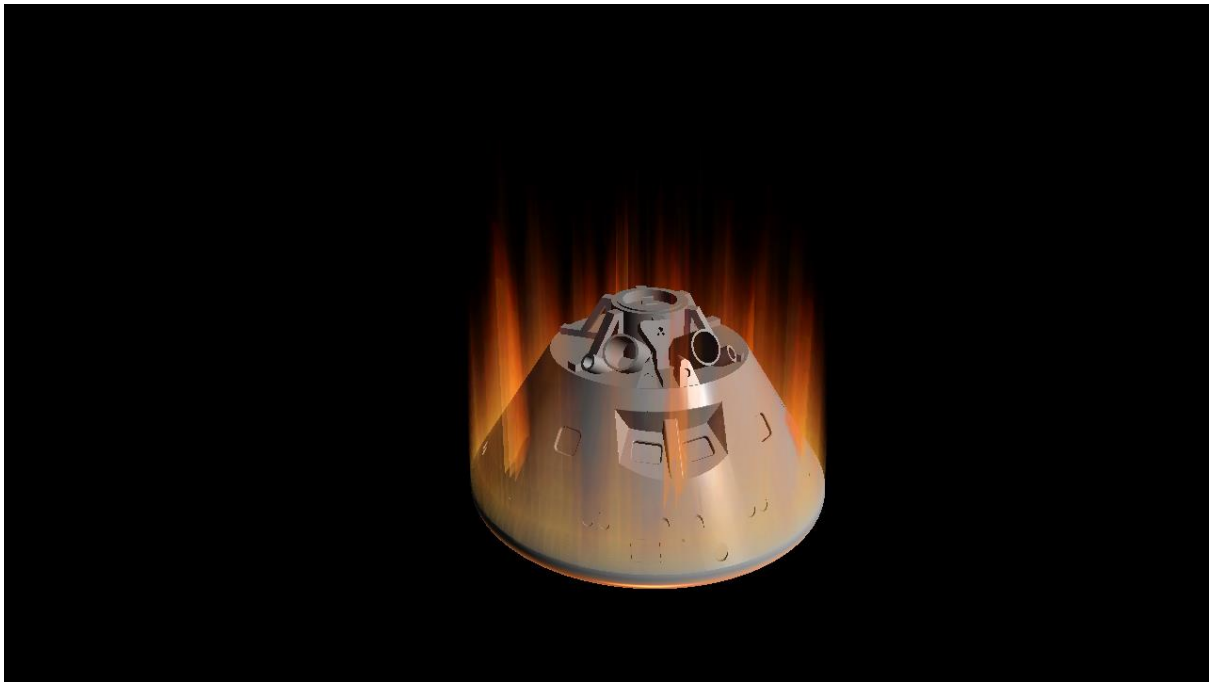
```
[maxvertexcount(6)]
void VelocityTrailerGS(triangle GS_INPUT vertex[3], inout TriangleStream<GS_DATA> triStream)
{
    //colors
    float3 brightCol = gEntryLightColorBright.rgb;
    float4 col1 = float4(brightCol, gEntryLightColorBright.a)*gVelocity;
    float4 col2 = lerp(gEntryLightColorBright, gEntryLightColorSecond, gColorModifier)*gVelocity;
    float4 col3 = gEntryLightColorSecond*gVelocity;

    float3 normVel = normalize(gVelocityDirection);

    for (int i = 0; i < 3; i++)
    {
        float3 normal = normalize(vertex[i].normal);
        float velDot = -dot(normal, normVel);
        float occlusion = CalculateShadow(vertex[i].lPos, gVelocityShadowMap, gVelocityShadowBias);
        //determining length based on normal
        if (velDot >= 0 && occlusion>0.9f)
        {
            int j = (i + 1) % 3;

            //Create vertices on the edge itself
            CreateVertex(triStream, vertex[i].worldPos, col1);
            CreateVertex(triStream, vertex[j].worldPos, col1);

            //Create "stacked" quads that create a gradient which simulates heat dissipation
            CreateVertex(triStream, vertex[i].worldPos + (gVelocity * gVelMultiplier * occlusion * velDot * normVel), col2);
            CreateVertex(triStream, vertex[j].worldPos + (gVelocity * gVelMultiplier * occlusion * velDot * normVel), col2);
            CreateVertex(triStream, vertex[i].worldPos + (gVelocity * gVelMultiplier * occlusion * velDot * normVel * gColorModifier), col3);
            CreateVertex(triStream, vertex[j].worldPos + (gVelocity * gVelMultiplier * occlusion * velDot * normVel * gColorModifier), col3);
        }
    }
}
```

This is what the effect looks like in the engine.

This is starting to look like the desired effect, however, we need to add a random length to the trail, so it doesn't look so uniform, and we need to change that length periodically, to "animate" the fire.

To achieve this, I use a perlin noise texture to make it random. For every vertex, I sample the normalized vertex position, and use the x and y coordinates of those as the uv coordinates in my sampler. This makes the fire more random at the tips, making it seem more like fire. However, this doesn't include the animation effect. To achieve that affect, I add a variable which I sum up on top of my UV coordinates. In the scene, I add a small value to this variable in every update tick, making the texture wrap around itself constantly, like a treadmill. This makes the fire jump up and down, animating it. Due to the limitations of a PDF, I cannot show the animation, however, here's how I did it with a picture of the variation in height.

```
float GetNoiseLength(float2 uv)
{
    float lengthVar = gNoiseMap.SampleLevel(gTextureSampler, uv, 0);
    return lengthVar * gLengthMultiplier;
}
```



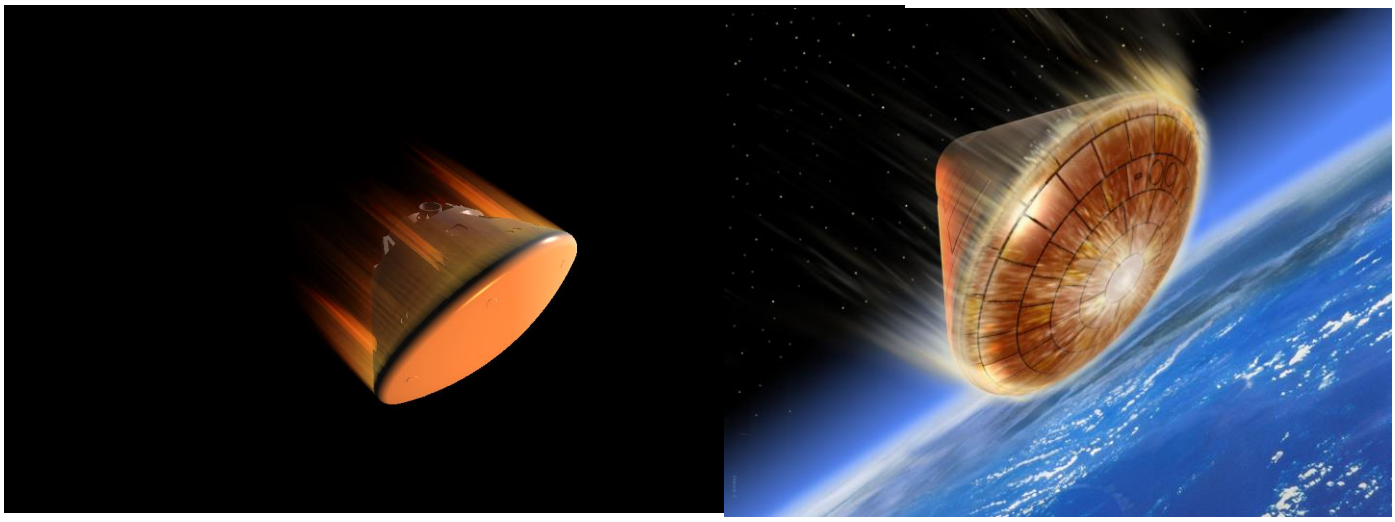
With the animation added, that concludes the geometry pass of the shader.

Conclusion

This concludes the shader. What I like about this shader is that you can expand on it a ton. You can add more pixel shaders, like normal maps or environment maps, but you can also expand on the geometry shader. I think the shader at its current state forms a solid base, but improvements can be made. For example, sometimes some of the fire still goes through the model, this is something that can be fixed with more advanced math, you can also add a variable to make the trail bounce off the front first, sideways, instead of immediately going along the airstream. I think the random height of the fire could also use some tweaking, as right now it seems obvious that it's pseudo-random.

Overall this was a very pleasant shader to work on. It was complicated, but this made it a technical challenge, but a satisfying challenge to complete. The biggest problem I ran into is finding a way to detect the edge of the model, but the developers of KSP gave me some ideas to solve this.

Here's a side-by-side comparison



Sources

https://nasa3d.arc.nasa.gov/detail/orion-capsule

http://www.spaceflightinsider.com/missions/human-spaceflight/nasa-releases-video-orions-fiery-reentry-earths-atmosphere/

https://www.youtube.com/watch?v=-FsvCAc4iTU

http://www.esa.int/var/esa/storage/images/esa_multimedia/images/2000/08/atmospheric_re-entry_demonstrator_-_artist_s_impression/9221109-5-eng-GB/Atmospheric_Re-entry_Demonstrator_-_artist_s_impression.jpg

https://upload.wikimedia.org/wikipedia/commons/thumb/8/87/Apollo_cm.jpg/220px-Apollo_cm.jpg

https://youtu.be/mXTxQko-JH0?t=24m30s

https://www.youtube.com/watch?v=LMI8PVwEf88