Atmospheric Reentry Geometry Shader

Leah Lindner

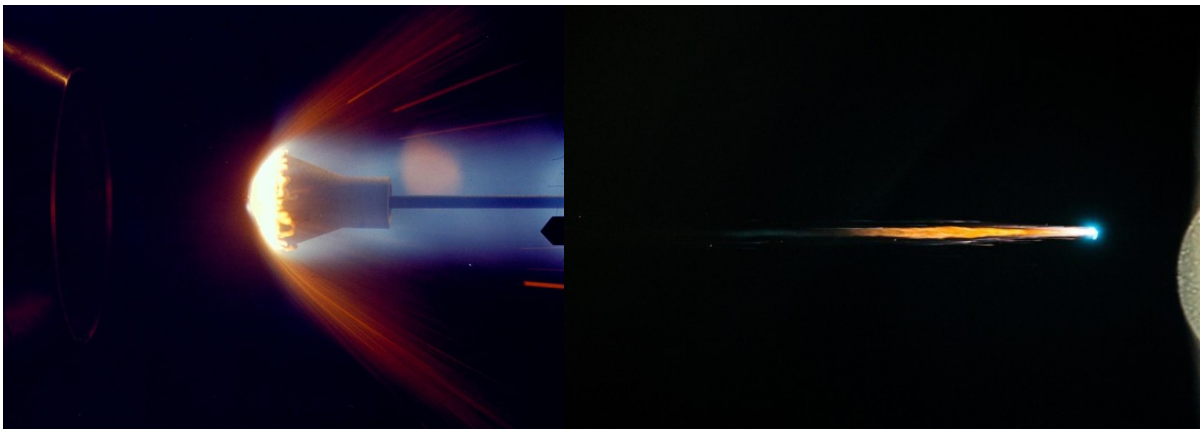
Introduction

In order to simulate the effect of an object – be it an asteroid, UFO or spacecraft – entering the atmosphere of a planet, I created a geometry shader effect that can work with variable geometry to visually approximate the plasma stream coming off of the object. In this paper I will present my approach to the subject. In real life this effect is created by the friction from air particles bouncing off the object travelling at high velocities (usually many kilometers every second), causing such an extreme head that a thick stream of plasma will form around the object and ablate any materials on the surface.

It is relevant in modern spaceflight (for instance Apollo and Soyuz space capsules, the Space Shuttle, and the Falcon 9 landing rocket), since real spacecraft use the friction to reduce most of the speed to get from orbital to landing velocity. As a visualization would be useful in spaceflight simulations and video games.

The main problem with the effect though is that no one really knows what it would look like from close up, but that is exactly what needs to be determined if the visualization is supposed to show the outside of the spacecraft. The only real reference available are photos taken from very far away with low detail and footage taken from the interior of reentering spacecraft.

Additionally computer and wind tunnel simulations have been made, and artistic visualizations exist.



The effect has notably been recreated in the game Kerbal Space Program and the movie Gravity, while the latter used fluid dynamics which rely on expensive calculations that are not possible in real time.

Nethertheless I found that from the reference I have the effect can be split into four different parts:

Effect Breakdown

Particles

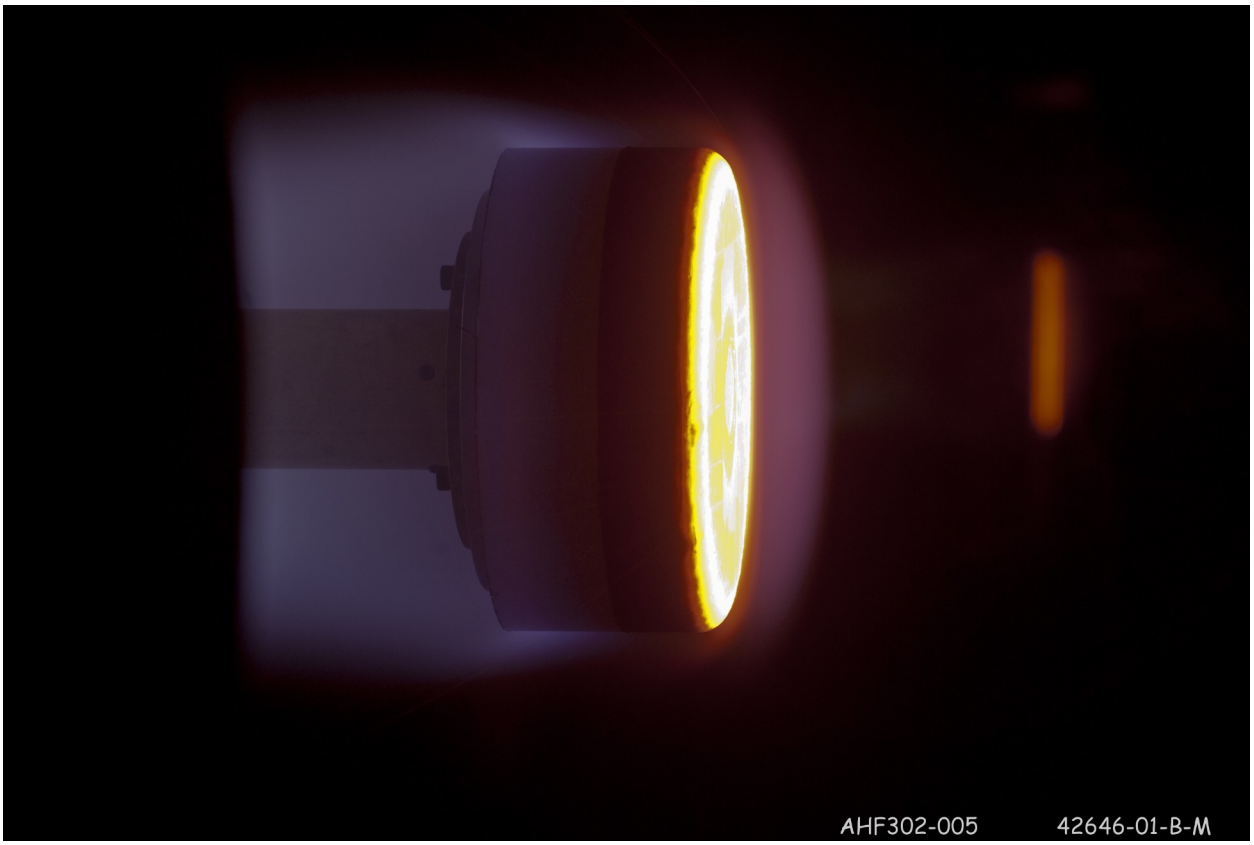
The first is a long stream of particles behind the object that changes with the hue and intensity with the particles lifetime. This effect can be replicated with particle systems in most game engines and will therefore not be the main focus of the geometry shader, although it needs to be ensured that the colours match with the colours of the shader.

The second effect is another one that can be achieved with a particle system which would simulate small chunks of ablated material flying off of the surface of the object, which I will also ignore in this paper for simplicity, although I would recommend implementing it for a full simulation.

Geometry

The third effect is the first that can be created with a geometry shader: At the highest velocities air molecules bounce off directly in front of the object and clutter the space in front of it, while creating plasma that is more hot than the rest and therefore appears blue. To visualize that the responsible pass of the shader will copy the object foreward along its movement direction, giving it a blue color and

intensity depending on the angle.



The fourth and most notable effect are the air molecules streaming around the object. Those streams may be rippling randomly at high velocities and will seem to be encapsulating the object along its movement vector, except at the back, from where the object would be visible if not for the long particle stream behind it. This part of the effect is like the previous one rather bright and may range in color from blue to orange to red. The effect is replicated by extruding geometry along the reverse velocity vector and lighting the object directionally from infront of it.

Shader Implementation

For my shader implementation I separate the shader into three passes:

```
//TECHNIQUES
//**********
technique11 TechBlinn
{
    pass p0
    {
        SetRasterizerState(gRS_NoCulling);
        SetVertexShader(CompileShader(vs_4_0, MainVS()));
        SetPixelShader(CompileShader(ps_4_0, MainPSBlinn()));
    }
    pass p1
    {
        SetRasterizerState(gRS_NoCulling);
        SetDepthStencilState(DisableDepthWriting, 0);
        SetBlendState(AlphaBlending, float4( 0.0f, 0.0f, 0.0f, 0.0f ), 0xFFFFFFFF);

        SetVertexShader(CompileShader(vs_4_0, EffectVS()));
        SetGeometryShader(CompileShader(gs_4_0, EffectGS()));
        SetPixelShader(CompileShader(ps_4_0, EffectPSBlinn()));
    }
    pass p2
    {
        SetRasterizerState(gRS_NoCulling);
        SetDepthStencilState(DisableDepthWriting, 0);
        SetBlendState(AlphaBlending, float4( 0.0f, 0.0f, 0.0f, 0.0f ), 0xFFFFFFFF);

        SetVertexShader(CompileShader(vs_4_0, EffectVS2()));
        SetGeometryShader(CompileShader(gs_4_0, EffectGS2()));
        SetPixelShader(CompileShader(ps_4_0, EffectPSBlinn()));
    }
}
```

First pass

The first pass renders the object normally with the addition of the extra light which is placed along the velocity vector of the object in order to visualize the frontal part of the encapsulation of the object, since I want to avoid extruding long geometry through the middle of its surfaces because it poke all the way though the object. The brightness of the light is affected by the overall strength of the effect, so it is multiplied with the effect intensity.

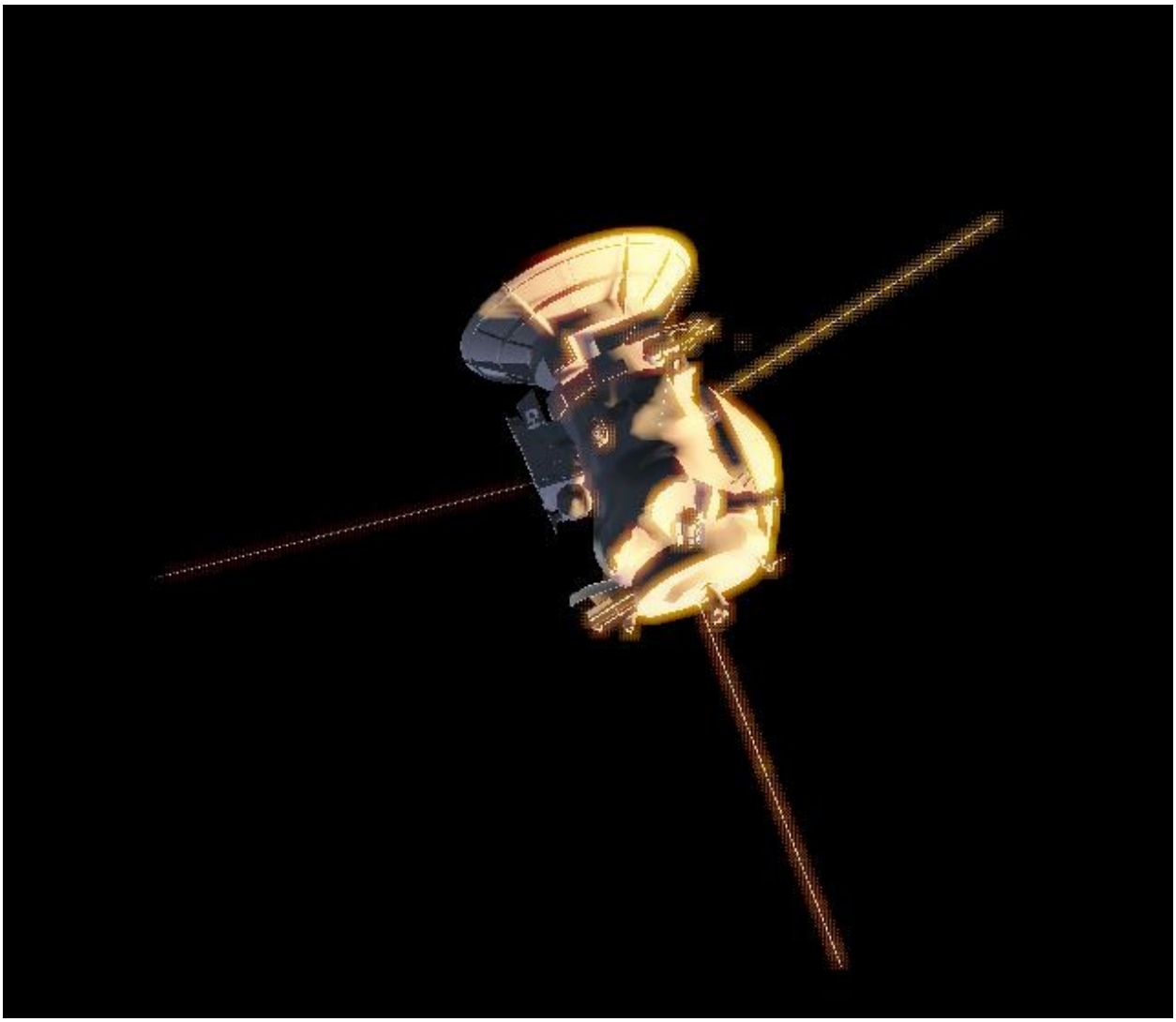
```
//Lighting
float shadowFactor = CalculateShadow(input.lpos, gShadowMap, gShadowMapBias);
float moveShadow = CalculateShadow(input.MoveLightPos, gMoveShadowMap, gMoveShadowBias);

float3 ambient = gColorAmbient*gAmbientIntensity;
float3 lightingSun = CalculateLighting(normal, gLightDirection, viewDirection, input.TexCoord)*gSunStrength;
float3 planetShine = CalculatePlanetShine(normal, float3(0, 1, 0), gAtmosphereColor.rgb*0.1f, input.TexCoord);
float3 reentryLighting = CalculateLighting(normal, gMoveDir, viewDirection, input.TexCoord)*gReentryLightColor.rgb*gReentryLightColor.a;
return float4(environment+ambient+shadowFactor*lightingSun+planetShine+reentryLighting*moveShadow*gReentryStrength*gMoveVel/4000, 1);
```

This light is also rendered with its own shadow map, which is used both to occlude in the pixel shader and the geometry shader later.

Additionally I create some artificial lighting to simulate the light shining off a planet, but that is only for visual enhancement and does not affect the reentry effect.

Here the result can be seen:



Second Pass

This is the main bulk of the work, here the geometry is extruded along the velocity vector.

To do this, multiple things need to be taken into consideration:

- Occlusion: Only surfaces directly exposed to the airstream are extruded. To do this the same shadow map as mentioned earlier is used in order to prevent unexposed surfaces from creating plasma. Additionally if the angle of the normal and the movement vector is larger than 90° it

also is not extruded, this is determined with a dot product

```
//movement "Shadow" occlusion
float3 occlusion = float3(CalculateShadow(vertex[0][0].LightPosition, gMoveShadowMap, gMoveShadowBias),
                          CalculateShadow(vertex[1][0].LightPosition, gMoveShadowMap, gMoveShadowBias),
                          CalculateShadow(vertex[2][0].LightPosition, gMoveShadowMap, gMoveShadowBias));
```

- A consistent stream regardless of geometry density: In my first implementation I simply treated the extrusion on a vertex by vertex basis, extruding a triangle strip from every point, but since models for realtime applications usually do not have consistent geometry density this would result in large gaps in some places and extremely dense plasma in others. So instead the input geometry needs to be a triangle in order to extrude triangle strips along the edges of the object.

```
[maxvertexcount(6)]
void EffectGS(triangle GS_INPUT vertex[3][1], inout TriangleStream<GS_DATA> triStream)
{
```

- Density: As mentioned earlier I only want to extrude the geometry if its on the edge in order to prevent it poking through the spacecraft. To do that I use the same dot product inverted and raise it to a power as a length multiplier, so that only the edges on the edge of the model actually extrude long streaks of plasma

```
float velDot = -dot(normal, nMove);
//determining length based on normal
if(velDot >= 0 && occlusion[i]>0.9f)
{
    int j = (i+1)%3;

    float3 nCenter = normalize(vertex[i][0].CenterVec);
    float3 nSize = normalize(cross(nMove, nCenter));

    velDot = pow(1 - velDot, 3);//assuming velDot larger 0
    float3 length_i = lengthVar[i]*velDot*nMove;
    float3 length_j = lengthVar[j]*velDot*nMove;
```

- Randomness: In order to make a random length I use a noise texture which is treated like an animated spritesheet. The subimage index of this frame and the next frame are passed along with the texture and a float that indicates the interpolation value between the new frames. The calculation of the indices and interpolation value is handled by the software, not in the shader. The indices are used to calculate the texture coordinates for two texture lookups. The values found by the lookups are interpolated between. To make it look a little more smooth, and added to the length of the extrusion

```
float GetNoiseLength(float2 tc, float2 tcN)
{
    float lengthVar = gNoiseMap.SampleLevel(gDiffuseSampler, tc, 0)[gSubImageChannel];
    float lengthVarNext = gNoiseMap.SampleLevel(gDiffuseSampler, tcN, 0)[gSubImageChannelN];
    return lerp(lengthVarNext, lengthVar, gNoiseLerp);
}
```

- Color: The color will go from red at the end of the streak over orange to a value between yellow and blue depending on the intensity of the effect. The alpha is also determined by the intensity, and additive blending is used for the streaks without writing to the depth buffer, just like with

particle systems that use textures on billboarded planes in order to prevent artifacts.

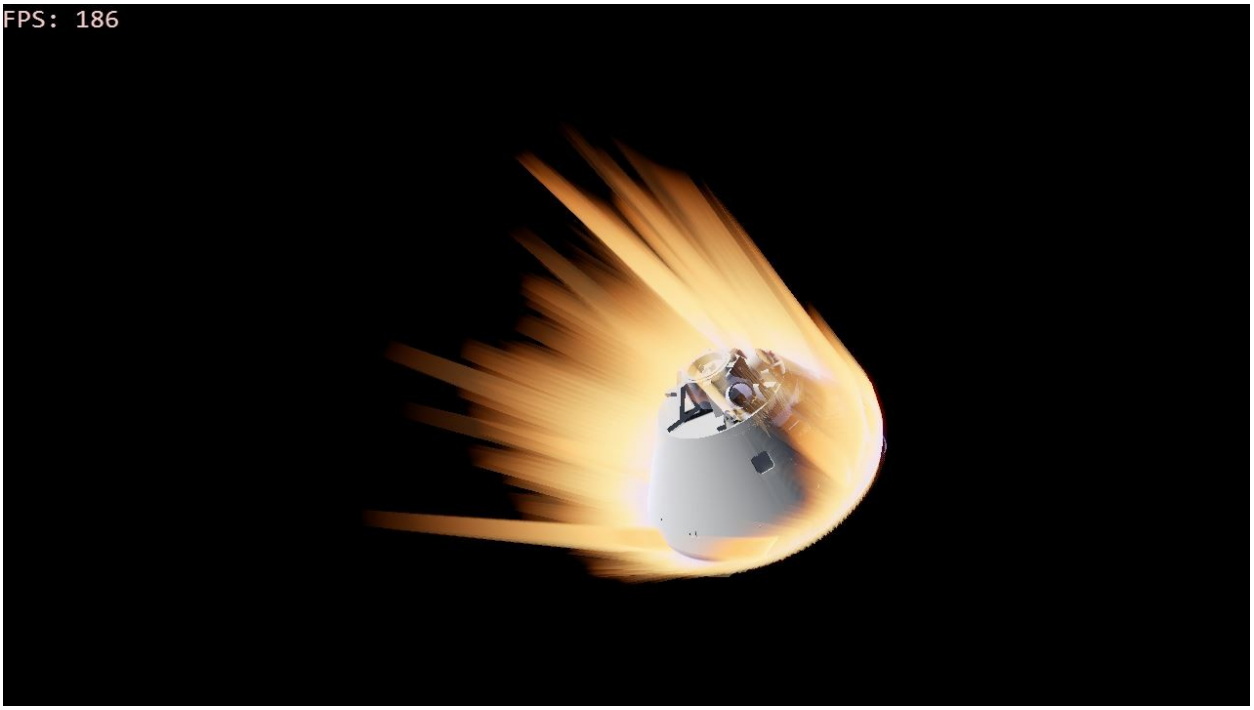
```
//colors
float strength = gMoveVel/4000;
float3 brightCol = lerp(gReentryLightColorBright.rgb, gBounceColor.rgb, saturate(strength-gBounceColLerpOffset));
float4 col1 = float4(brightCol, gReentryLightColorBright.a)*strength;
float4 col2 = lerp(gReentryLightColorBright, gReentryLightColorRed, gIncrement)*strength;
float4 col3 = gReentryLightColorRed*strength;
```

- Geometry Construction: I found that the effect works quite well by giving every strip 6 triangles (creating three points along the streak, one at the origin(yellow/blue), one somewhere along the middle(orange), and one at the end (pale red)). Using the middle vertices the strip is slightly curved, which improves the quality of the effect without too much performance impact. Since a lot of streaks are rendered using a lot more geometry wont significantly enhance the level of detail of the effect.

```
float3 outward = -cross(nMove, nSize)*gIncrement*lengthVar[i]*velDot;
float3 sizeEnd = nSize*gTaperSize;
float3 sizeTop = nSize*lerp(1, gTaperSize, gIncrement);

float3 end_j =  sizeEnd +length_j            + (outward);
float3 end_i = -sizeEnd +length_i            + (outward);
float3 top_j =  sizeTop +length_j*gIncrement + (outward*curve);
float3 top_i = -sizeTop +length_i*gIncrement + (outward*curve);
//Create Geometry (Trianglestrip) alternating between this and its adjacent edge
CreateVertex(triStream, vertex[i][0].Position,          col1);
CreateVertex(triStream, vertex[j][0].Position,          col1);
CreateVertex(triStream, vertex[i][0].Position + top_i,  col2);
CreateVertex(triStream, vertex[j][0].Position + top_j,  col2);
CreateVertex(triStream, vertex[i][0].Position + end_i,  col3);
CreateVertex(triStream, vertex[j][0].Position + end_j,  col3);
```

Here the result of this can be seen:



Third Pass:

This is already looking rather good, but it is still missing the effect of blue particles bouncing off at the front, so that is what the third pass takes care of.

It determines the geometry that needs to be copied forward based on the shadow map and the dot product between the normal and the movement vector, and is also raised to a power, which also helps determine the intensity of its blue color, to ensure that only faces with an angle very close to the velocity vector will produce this effect.

```
[maxvertexcount(3)]
void EffectGS2(triangle GS_INPUT2 vertex[3][1], inout TriangleStream<GS_DATA> triStream)
{
    float strength = gMoveVel/4000;
    float3 occlusion = float3(CalculateShadow(vertex[0][0].LightPosition, gMoveShadowMap, gMoveShadowBias),
                              CalculateShadow(vertex[1][0].LightPosition, gMoveShadowMap, gMoveShadowBias),
                              CalculateShadow(vertex[2][0].LightPosition, gMoveShadowMap, gMoveShadowBias));

    float3 nMove = normalize(gMoveDir);

    for(int i = 0; i < 3; i++)
    {
        float3 normal = normalize(vertex[i][0].Normal);
        float velDot = dot(normal, nMove);
        velDot = pow(velDot, 3);
        if(occlusion[i]>0.9f && velDot>0)
        {

            float alpha = -velDot*occlusion[i]*strength;;
            float4 col = gBounceColor*alpha;
            float3 offset = alpha * gBounceOffset * normal;

            CreateVertex(triStream, vertex[i][0].Position+offset, col);
        }
    }
}
```

And here is the result of all the passes combined:

With some visual enhancements this is the look I managed to achieve:

Conclusion:

I find the shader created in this project quite satisfying, but I think there are one or two things that could be done to improve it. Notably it could take into consideration velocity vs air pressure to determine the strengths of different effects. For instance at a high velocity but low air pressure the streaks would be extremely long but the effect itself rather weak in terms of blending and would be blue to purple color wise, then at a high air pressure but low speed (later in the reentry) the streaks are not particularly long but rather hot, with the color more orange-ish, and at even lower speeds the color could be changed to white in order to represent mach effects (as is done in Kerbal Space Program).

Something that also could be done is blending the models diffuse texture between an intact and an ablated texture that looks more damaged, but I didn't use any diffuse texturing in this demo.

Lastly the particles could be used to enhance the visuals even more.

Regardless the current state of the shader should be well usable for any sort of spacesim game or program and I hope that this paper has helped explain how it can be constructed.

Sources:

https://www.youtube.com/watch?v=esMZjRtol5c

https://youtu.be/3nBo87Ne4Ss?t=31s

http://farm4.staticflickr.com/3063/2360738075_931822b24f.jpg

https://youtu.be/Cf_-g3UWQ04?t=1m23s

https://www.youtube.com/watch?v=MtWzuZ6WZ8E&nohtml5=False

https://youtu.be/KyZqSWWKmHQ?t=4m2s

https://youtu.be/mXTxQko-JH0?t=24m31s (5 minutes)

http://www.nasa.gov