Rebuild shader

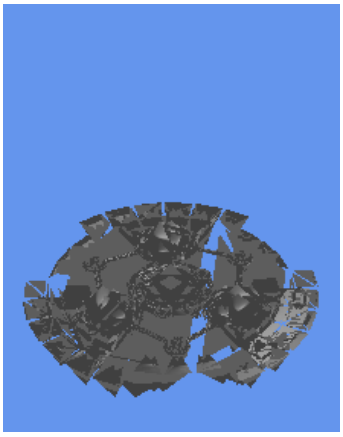
Introduction and uses

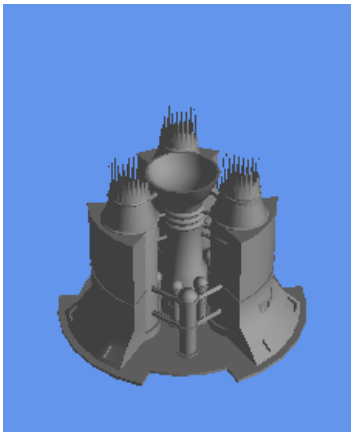
The rebuild shader is a HLSL shader that can be used as an explosion effect and as a reconstruction effect. The main purpose was to have a shader that could 'build' meshes when I open a scene in a game.

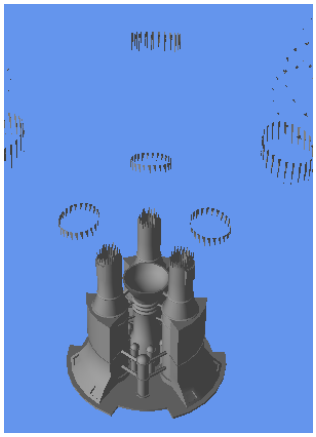
It can be used to draw distant meshes that would pop-in in the scene. Instead of just making them appear out of nowhere, you apply this shader and they assemble themselves.

Another way of implementing this shader is by using its explosion effect when, for example, a building gets hit by an explosive projectile. You can set the impact position and the debris will be pushed by a force from that location.

A last and a little bit dodgy implementation is to set the explosion height very high. Then let the time pass reversed. The mesh will appear to rise from a 2D (top-down) mesh on the ground to the original 3D mesh.







Inspiration and initialization

I wanted to make a shader that appeared to create something. The core idea was to scatter every part of the mesh on the ground and letting it fly to its destination. A bit like the Sandman from the movie Spiderman 2. So I thought about elevating the parts up in the air first, to later let them fall in their correct spot. During coding I added a little bit more to this idea; making the explosion force location editable or making debris fade in/out.

The road

I started by making an explosion shader. To both get my feet wet with a geometry shader and getting the basics of my shader. The explosion force was emitted from the center of the mesh, I wanted to look a bit like the explosions in Rollercoaster Tycoon. Next I added the *completion* variable. This is a zero-to-one value that represents the percent of the mesh that will not be affected by the explosion, from bottom to top. To make the debris look a little less flat a plane, I changed every triangle into a *chunk*. This made it look more like debris chunks and also give it a 3D touch.
I explain each important part with a bit of code in the next part.

Code Overview

There are the modifiable variables of the rebuild shader.

```
1.  Texture2D m_TextureDiffuse;     // the diffuse texture
2.  float m_LowerLimit = 0.0f;      // bottom, where the rebuilding starts at 0%
3.  float m_UpperLimit = 1.0f;      // top, where the rebuilding stops at 100%
4.  float m_Completion = 0.0f;      // the completion in %
5.  float m_SoftCompletion = 0.75f; // interpolation distance between completion and destro
    yed
6.
7.  float m_Time = 0.0f;            // the timeline of the shader
8.  float m_Gravity = -9.81f;       // gravitational force
9.  float m_BurstPower = 4.0f;      // explosive force of the chunks flying away
10. bool m_UseCustomExplosionPosition = false;          // enables/disables the use of a cu
    stom explosion force position
11. float3 m_CustomExplosionPosition = { 0, 0, 0 };     // center from which the explosive
    force emits
```

In the next bit of code I check if the triangle is affected by the explosion (burst). See the *topCompletionHeight* as a line that goes from the bottom to the top of the mesh. Everything below this line does not get affected by the explosion, all triangles above the line do. The *m_SoftCompletion* makes a linear interpolation between being affected or being unaffected by the burst. The value represents the range from the completion line to (downwards) the completion line's height minus the value of *m_SoftCompletion*.

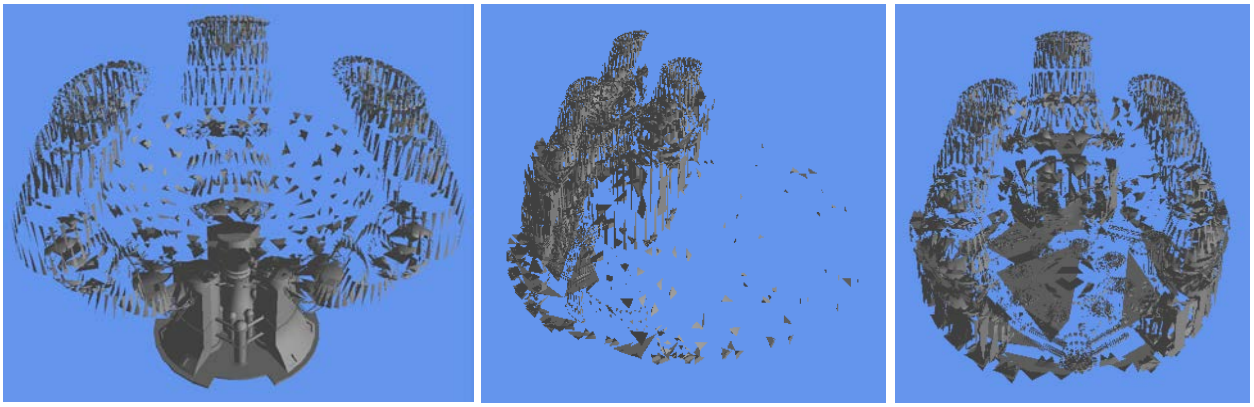
There's also a check to mark the triangle to create chunks or not (see further in this paper).

```
1.  // COMPLETION STUFF
2.  float topCompletionHeight = completionHeight + (m_SoftCompletion * clamp((completionHei
    ght * 10), 0, 1));
3.  float rebuildScale = clamp(((basePosition.y - completionHeight) / m_SoftCompletion), 0.
    0f, 1.0f);   // swapped topcompletion with cmopletionheight (locations in sums )
4.  // destructed
5.  if (basePosition.y >= topCompletionHeight)
6.  {
7.      // do nothing special
8.      time = time;
9.      createChunks = true;
10. }
11. // rebuild in progress
12. else if (basePosition.y < topCompletionHeight && basePosition.y > completionHeight)
13. {
14.     time = time * rebuildScale;
15.     createChunks = true;
16. }
17. // rebuilded
18. else
19. {
20.     // set time to 0, as if it never bursted
21.     time = 0;
22.     createChunks = false;
23. }
```

This part of the code applies the force to each triangle and pushes it away if it lies above the *m_Completion* variable. If the user sets a custom position for the explosion center the force will be emitted from that location, else the de default location will be the center of the mesh.

The further a triangle is away from the explosion center, the weaker the force of the explosion is to the triangle. There's a *maxDistance* variable to prevent parts from flying too far away.

```
1.  // BURST & BREAK PART
2.  float3 objectCenter = { 0, (m_LowerLimit + m_UpperLimit) / 2.0f, 0 };
3.  float3 explosionCenter = { 0, (m_LowerLimit + m_UpperLimit) / 2.0f, 0 };
4.  if (m_UseCustomExplosionPosition)
5.      explosionCenter = m_CustomExplosionPosition;
6.  float3 forceNormal = normalize(basePosition - explosionCenter);
7.  float forceDistance = abs(length(basePosition - explosionCenter));
8.  float maxDistance = forceDistance + ((m_UpperLimit - m_LowerLimit) * 1.25f);
9.  float force = clamp(abs(maxDistance - forceDistance)*time, 0, maxDistance) * m_BurstPow
    er;
10.
11. // add forces and such
12. VS_DATA newVertices[3]  = vertices;
13.
14. float3 gravityVelocity = {0,0,0};
15. gravityVelocity.y += m_Gravity * time * time;
16.
17. float3 forceVelocity = {0,0,0};
18. forceVelocity = forceNormal * force;
19. float3 offset = forceVelocity + gravityVelocity;
20.
21. for (int i=0; i<3; ++i)
22. {
23.     float3 newPosition = newVertices[i].Position + offset;
24.     if (newPosition.y < m_LowerLimit)
25.         newPosition.y = m_LowerLimit;
26.     newVertices[i].Position = newPosition;
27. }
```



This part creates the chunks. Chunks make the debris feel less flat. This function takes the average length of the triangle's sides and multiplies that value by three. Next it uses the opposite direction of the triangle normal to place a new vertex at the backside of that triangle. At the end of this function it simply draws three triangles from each side of the original triangle to the new vertex on the backside. Creating a little volume, the chunk. This is the setup at the start of the function.

```
1.  //setup base values
2.  float3 baseNormal= (vertices[0].Normal + vertices[1].Normal + vertices[2].Normal)/3.0f;
3.  float3 basePosition = (vertices[0].Position + vertices[1].Position + vertices[2].Positi
    on)/3.0f;
4.
5.  //get average length of the sides of the triangle
6.  float baseLength = abs((vertices[0].Position - vertices[1].Position) + (vertices[1].Pos
    ition - vertices[2].Position) + (vertices[2].Position - vertices[1].Position)) / 3.0f;
7.  //multiply with a scalar to get a more protruding chunk
8.  baseLength *= 3.f;
9.
10. float3 newNormal = baseNormal * -1;
11. float3 newPosition = basePosition + (newNormal*baseLength);
12.
```

Here we create one side of the new chunk. Do this three times for each new side of the chunk.

```
1.  if (createChunks)
2.  {
3.      // creade side
4.      CreateVertex(triStream,vertices[2].Position,vertices[2].Normal,vertices[2].TexCoord
    );
5.      CreateVertex(triStream,newPosition,newNormal,vertices[1].TexCoord);
6.      CreateVertex(triStream,vertices[0].Position +addPos,vertices[0].Normal,vertices[0].
    TexCoord);
```

References

- The shader we made in the labs, mainly the spiky shader – getting started
- Crash explosion effect (Rollercoaster Tycoon) – the basic explosion shader parts
- Sandman (Spiderman 2) – Chunks flying up and then falling in their places