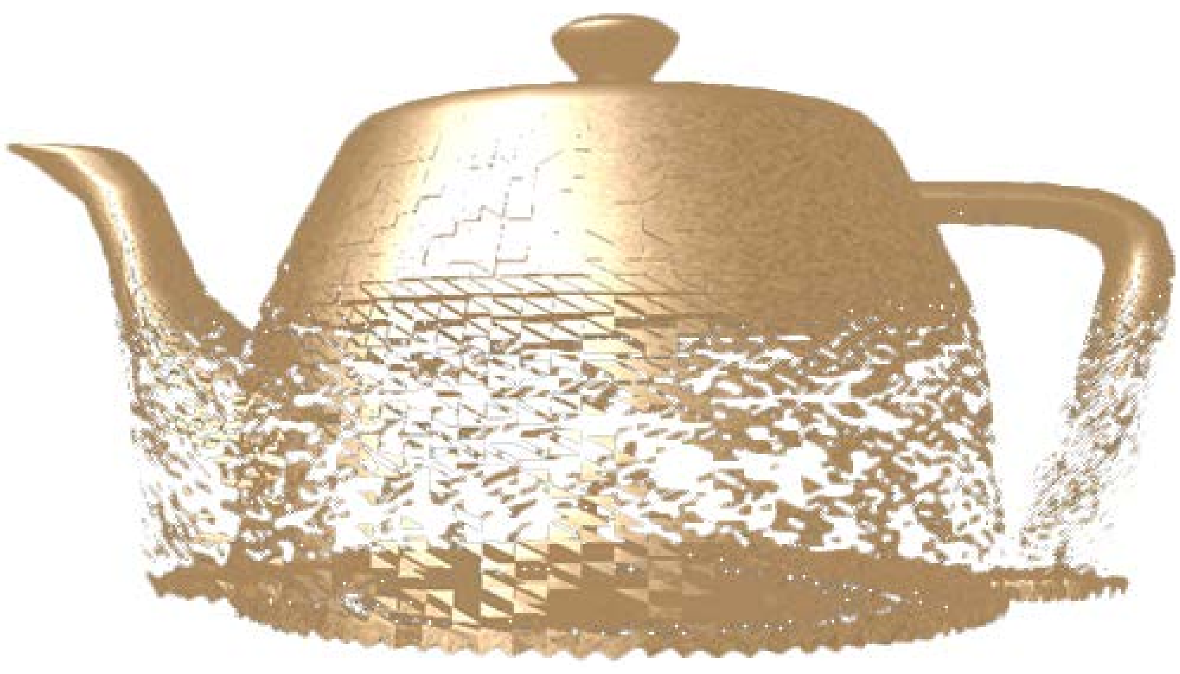
Geometry Shader: Dissolve Effect

Yosha Vandaele

[2017]



Introduction

The goal of this project is to create a shader that emulates the dissolving of a mesh from the bottom up as if it were made of sand. First we will subdivide triangles in order to have more and smaller particles to work with. The actual effect consists of a combination of a small explode translation and a gravity translation. Both translations are dependent on the initial height of the vertices, meaning that the lowest vertices will start to explode and fall first. This way it looks as if the object is dissolving starting from the bottom and the loss of support enables the above triangles to fall. In order to add some irregularity we use samples of a noise texture to influence the speed and distance of the translation for each vertex.

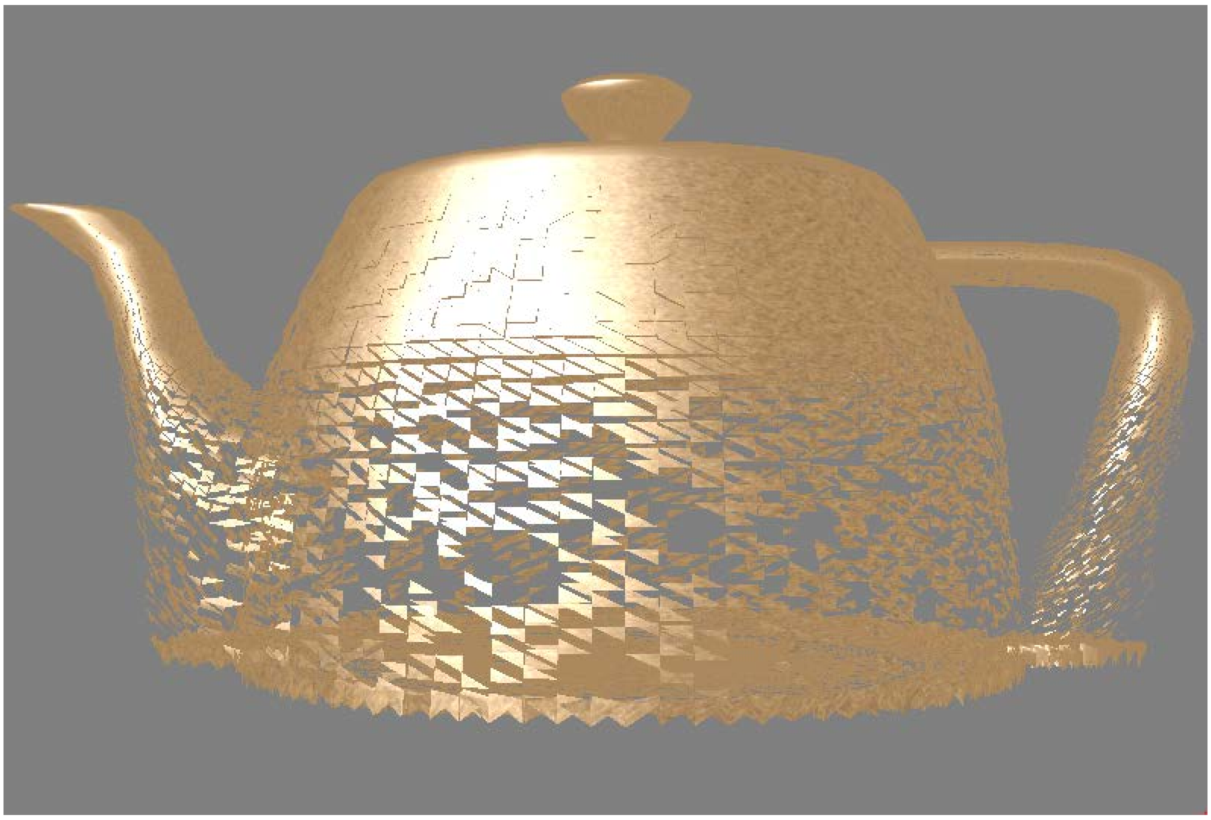


Figure 1: Dissolve Shader

Generating geometry: Subdivision

The first step is to subdivide the given triangles into more triangles. To do this we create a function that takes 3 vertices and outputs 12. This method can later on be used on the newly generated triangles to subdivide even further (from 3 to 48 vertices).

To do this I calculate the midpoint of each segment by taking the average position, normal and texture coordinate of the 2 points defining that segment. These 6 vertices are used to define 4 separate triangles, resulting in a 12 point array.

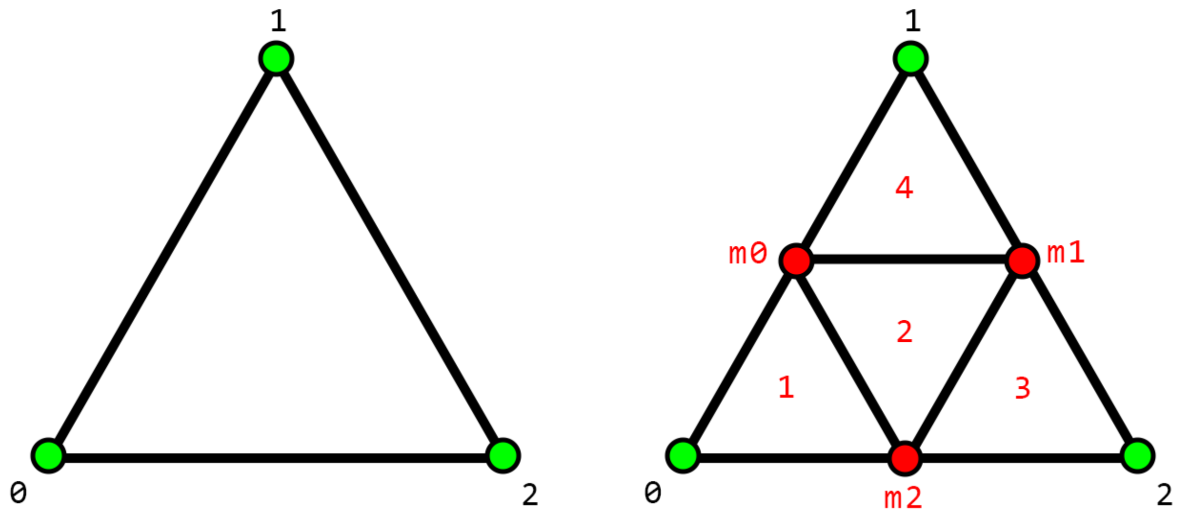


Figure 2: Subdivision of a triangle

```
/*----------------------------------------------------------------------*/
 void Subdivide(VS_DATA vertsIn[3], out VS_DATA vertsOut[12])
 {
    VS_DATA m[3];
    // Calculate the 3 midpoints of the triangle edges
    m[0].Position = (vertsIn[0].Position + vertsIn[1].Position) * 0.5f;
    m[1].Position = (vertsIn[1].Position + vertsIn[2].Position) * 0.5f;
    m[2].Position = (vertsIn[2].Position + vertsIn[0].Position) * 0.5f;
    // Calculate normals of the midpoints
    m[0].Normal = (vertsIn[0].Normal + vertsIn[1].Normal) * 0.5f;
    m[1].Normal = (vertsIn[1].Normal + vertsIn[2].Normal) * 0.5f;
    m[2].Normal = (vertsIn[2].Normal + vertsIn[0].Normal) * 0.5f;
    // Calculate texture coordinates of the midpoints
    m[0].TexCoord = (vertsIn[0].TexCoord + vertsIn[1].TexCoord) * 0.5f;
    m[1].TexCoord = (vertsIn[1].TexCoord + vertsIn[2].TexCoord) * 0.5f;
    m[2].TexCoord = (vertsIn[2].TexCoord + vertsIn[0].TexCoord) * 0.5f;
/*----------------------------------------------------------------------*/
```

Animating the added geometry over time

We will now use the newly generated geometry in a few separate steps to emulate the dissolve effect. First we will apply an explode effect to separate triangles slightly outwards from the main body. This makes it look as if the mesh loses cohesion while the triangles are starting to fall. Then we will emulate gravity, also starting from the bottom up and with a variable fall depth to generate the pile of remainders. In this final step we will try to make the triangles end up in such a way that they resemble a convincing pile.

Before moving on to the actual math we need to get some basic variables ready. First of all we want to be able to manipulate the speed of the effect so we multiply time with speed. To do calculations regarding gravity we get the positions of our three vertices translated to world space. We will also be using the average height of these three vertices in order control which triangles explode and fall first and how far they fall. To add some irregularity to the different aspects of this effect we will obtain a sample from a noise texture. Finally we will be using a variable for the average distance of a triangle from the center of the mesh which is used to affect the shape retention of the pile. For each of these effects we will loop through 3 of the newly generated vertices until all 12 vertices have been modified.

```
/*------------------------------------------------------------------------*/
// Scale time
float time = gTime * gSpeed;
// Get vertices in world space
float3 pointA = vertsOut[i].Position;
float3 pointB = vertsOut[i + 1].Position;
float3 pointC = vertsOut[i + 2].Position;

pointA = mul(pointA, (float3x3) gWorld);
pointB = mul(pointB, (float3x3) gWorld);
pointC = mul(pointC, (float3x3) gWorld);

// Noise Sample
float3 noiseSample = float3(0.0f, 0.0f, 0.0f);
noiseSample = gNoiseMap.SampleLevel(
                  samLinear, vertsOut[i].TexCoord, 1.0f) - gFloorHeight;

// Average height of the 3 vertices
float averageHeight = (pointA.y + pointB.y + pointC.y) / 3.0f;

// Average distance from center
float localX = vertsOut[i].Position.x;
float localZ = vertsOut[i].Position.z;
float outwardDistance = sqrt(localX * localX + localZ * localZ);
/*------------------------------------------------------------------------*/
```

Explode

To make it look more like the shape is crumbling or collapsing, part of the triangle movement consists of an explode effect.

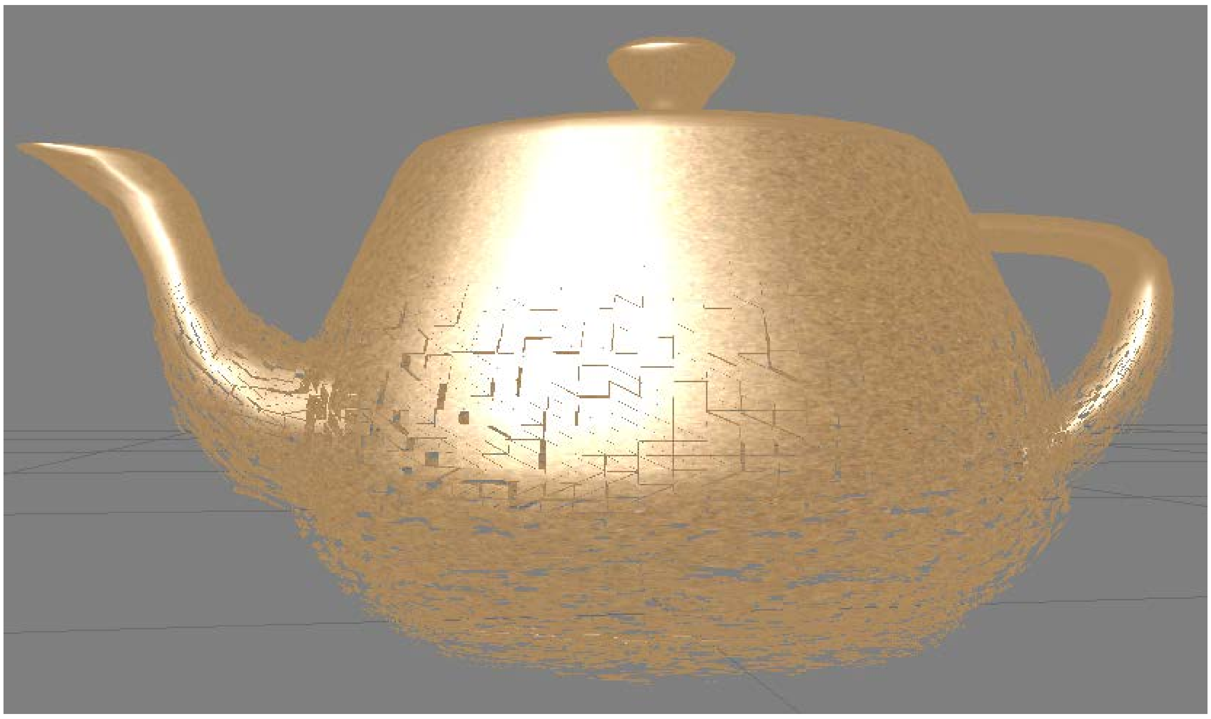


Figure 3: Explosion starts at the bottom

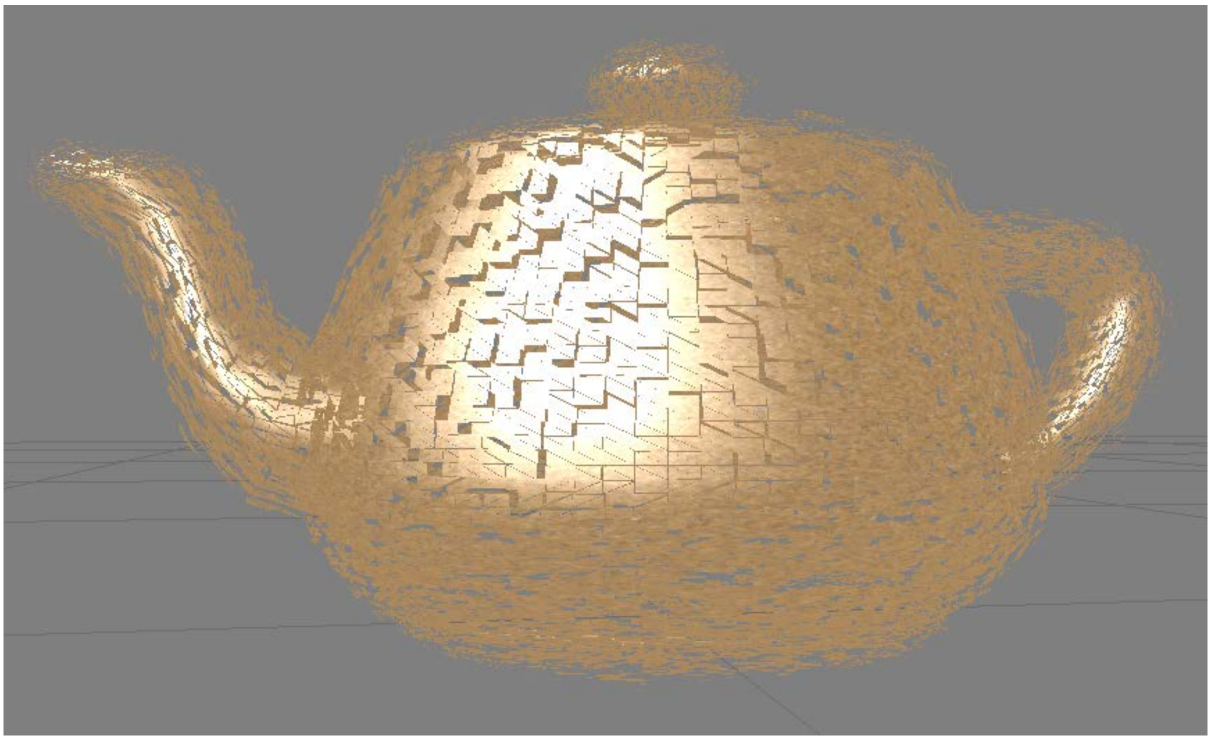


Figure 4: Explosion completed

In this stage every triangle gets pushed outwards over time in the direction of its own normal. The actual displacement is determined by the user through the variable *gExplodeDistance*. Additionally the explosion can be influenced by the height of the vertices to make the explosion start from the bottom. This value (*gExplodeHeightInfluence*) gets multiplied by the average height of the 3 vertices. The offset gets multiplied by the elapsed time minus that height influence, resulting in the lowest triangles being affected first. Finally we use a sample from the noise texture to introduce some irregularity to the explosion. This value gets multiplied with the variable *gExplodeNoiseImpact* to give the user more control over this noise factor.

```
/*----------------------------------------------------------------------*/
                            /* ---EXPLODE--- */
float3 explodeNormal =
(vertsOut[i].Normal + vertsOut[i + 1].Normal + vertsOut[i + 2].Normal) / 3;

explodeNormal = normalize(explodeNormal);
float3 explodeOffset = explodeNormal *gExplodeDistance;

float explodeHeightInfluence = averageHeight * gExplodeHeightInfluence;

pointA += explodeOffset
            * saturate(max(0, ((gTime – explodeHeightInfluence))))
            * noiseSample.y * gExplodeNoiseImpact;
pointB += explodeOffset
            * saturate(max(0, ((gTime – explodeHeightInfluence))))
            * noiseSample.y * gExplodeNoiseImpact;
pointC += explodeOffset
            * saturate(max(0, ((gTime – explodeHeightInfluence))))
            * noiseSample.y * gExplodeNoiseImpact;
/*----------------------------------------------------------------------*/
```

Gravity

To emulate gravity we apply a formula which is loosely based on the free fall formula used in physics. First we calculate the offset based on the elapsed time (squared) and the average height of the three vertices. The average height gets multiplied by *gHeightDelay*, with which we control the difference in fall timing between higher and lower triangles. Because we subtract these height values from the time we use max to ensure only values larger than 0 are outputted.

Next up the noise that inflicts variance to the fall effect is calculated from the user inputted texture. We take the average of the sample and 1 to prevent too large a disparity as well as zero values. These zero values would otherwise result in certain vertices never moving at all.

We will ignore the pile generation for now and move on to the final step in the gravity code which is to subtract the offset from the y-value of the vertex at hand. Subsequently the resulting height is clamped between the original height and the floor height (*gFloorHeight*) to ensure the particles “collide” with the floor in order to form a pile.

```
/*----------------------------------------------------------------------*/
/* POINT A */
// calculate a factor related to time and height to make the lowest points
// fall at a certain rate faster than the ones above
float heightTimeOffset =
      max(0, time * time - averageHeight * gHeightDelay);
// calculate noise factor
float noise = (1 + noiseSample.z) / 2;
// calculate factor that determines the fall depth
float pileHeightFactor =
      pointA.y - pow(pointB.y / 4, 2) + outwardDistance * gPileShape;
pileHeightFactor = max(0, pileHeightFactor);
pileHeightFactor *= gPileFlatness;
// calculate the actual offset with the above factors and the gHeightDelay
float offset = heightTimeOffset * noise;
offset = pow(offset, 2);
// clamp the offset to the pile height
offset = clamp(offset, 0, pileHeightFactor);
pointA.y -= offset;
pointA.y = max(gFloorHeight, gFloorHeight + pointA.y);

/*----------------------------------------------------------------------*/
```

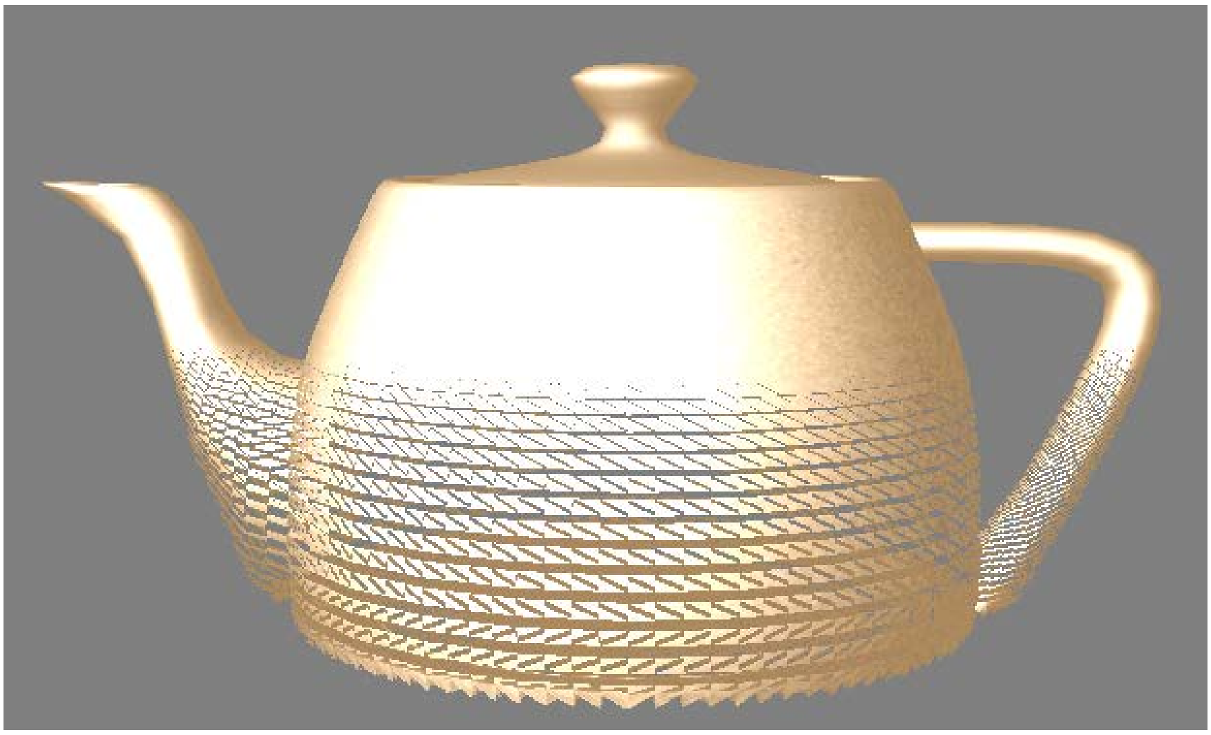


Figure 5: Emulating gravity [gHeightDelay = 3]

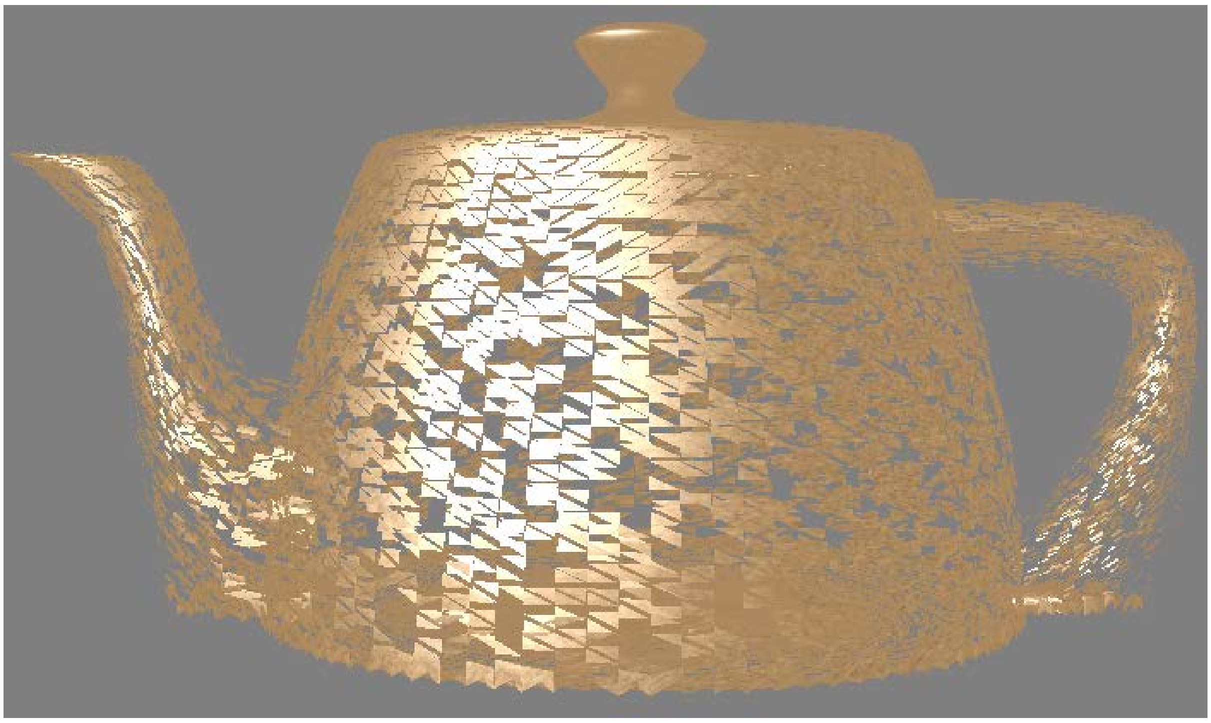


Figure 6: Emulating gravity [gHeightDelay = 1]

Pile Generation

In order to create a convincing pile the fall depth gets limited to an amount determined by the original height of the vertex and a global variable *gPileFlatness* which ranges from 0 to 1. Another factor that affects the pile height and shape is the distance from the vertex to the center point of the mesh. For simple shapes one can assume the highest part of the pile will be in the center because stuff tends to collapse outward. The variable *gPileShape* offers a measure of control for the user regarding the final shape of the pile. Before subtracting the offset from the original height we clamp its value to the resulting pile height.

The explode effect and shape distortion during the gravity effect make it so the resulting geometry is quite incoherent and contains a lot of gaps. To make it look more like a complete shape we pick one of the vertices of each triangle and make it drop all the way to the floor. However we apply the same formulas for the offset as we did for the other two vertices to prevent stretching and difference in fall speed during the gravity effect. Eventually we use this offset divided by the initial height to do a linear interpolation between that height and the floor level.

```
/*----------------------------------------------------------------------*/
/* POINT B */
offset = heightTimeOffset * noise;
offset = pow(offset, 2);
offset = clamp(offset, 0, pointB.y);
// lerp point B to floor height in order to connect each triangle to the
// floor
pointB.y = lerp(pointB.y, gFloorHeight, offset / pointB.y);
pointB.y = max(gFloorHeight, pointB.y);
/*----------------------------------------------------------------------*/
```

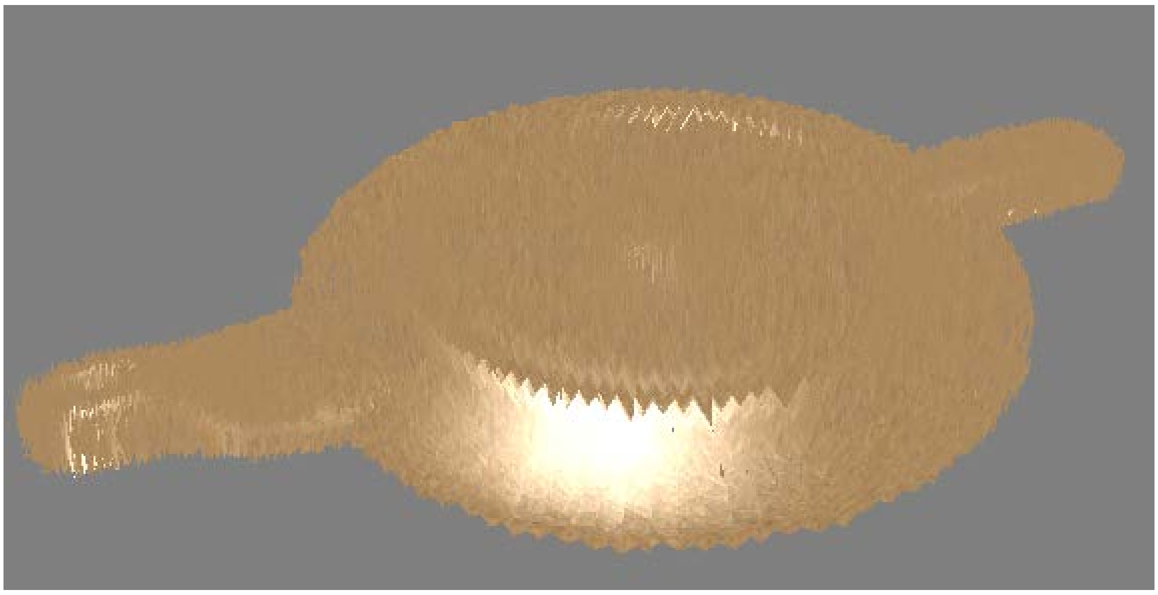


Figure 7: Pile [gPileShape = 0; gPileFlatness = 0.9]

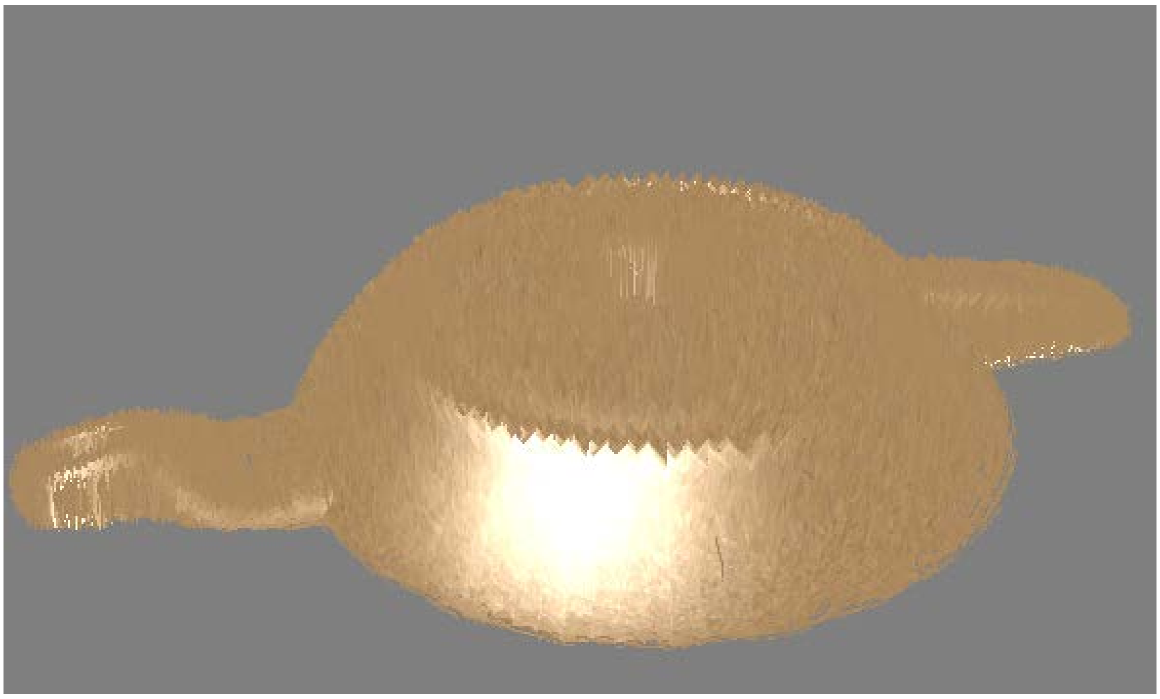


Figure 8: Pile [gPileShape = 0.1; gPileFlatness = 0.75]

References

https://www.youtube.com/watch?v=I8II3df8Tt0

https://www.gamedev.net/topic/635147-subdividing-triangles-with-the-geometry-shader/

http://diary.conewars.com/vertex-displacement-shader/