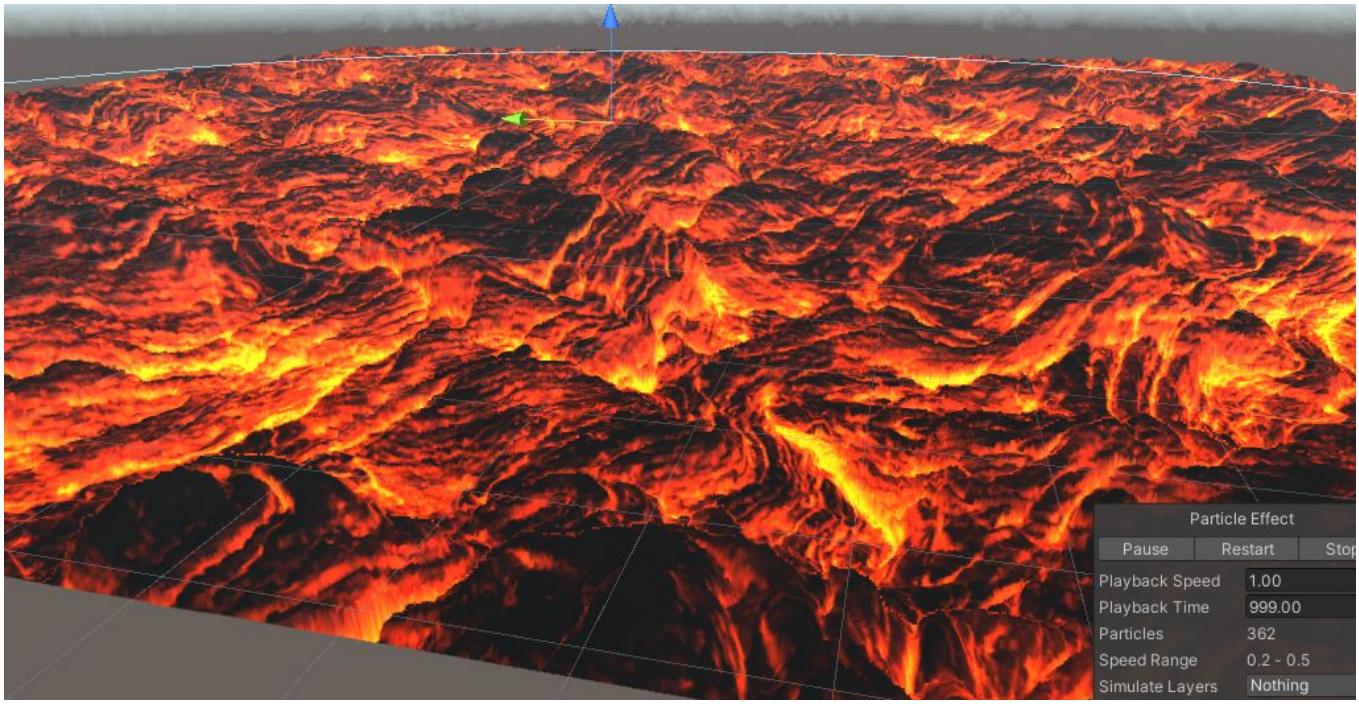


Dynamic Lava Shader

Game Lab - Raymond Becking 500785471

Moodboard: https://app.milanote.com/1JeF4m1v9QV5e9

Parallax Occlusion Mapping

My idea for this shader was to make a lava shader with proper normal/height(parallax) mapping for realism and depth. In my final product I used the following techniques: Parallax Occlusion Mapping, Emission map, Distortion Post Processing effect, Flowmap and Tiling(for POM).

Parallax map

```
void surf (Input IN, inout SurfaceOutputStandard o)
{
// Displacement
float heightTex = tex2D(_HeightTex, IN.uv_HeightTex).r;
float2 parallaxOffset = ParallaxOffset(heightTex, _Parallax, IN.viewDir);

fixed4 c = tex2D (_MainTex, IN.uv_MainTex + parallaxOffset) * _Color;

//Normal mapping
o.Normal = UnpackNormal(tex2D(_Normal, IN.uv_Normal + parallaxOffset));

o.Albedo = c.rgb;
}
```

To create a parallax shader I started on creating a surface shader.
The parallax effect based on the heightmap combined with a normal map gives an alright looking effect. heightTex contains the height of the texture(of a single pixel) based on a grayscale image. The height is then used to calculate the parallax offset using a built-in unity parallax function from UnityCG.cginc. The calculated offset is used to adjust the uv of the main texture & the normal texture. In my case the parallaxOffset for the MainTex is not actually necessary since my main texture does not have a lot of detail, and is mostly a uniform color. But since some do, I included it in case I want to reuse this shader. To implement the ParallaxOffset function I followed an article by Alisavakis(2019).

```
// Calculates UV offset for parallax bump mapping
inline float2 ParallaxOffset( half h, half height, half3 viewDir )
{
    h = h * height - height/2.0;
    float3 v = normalize(viewDir);
    v.z += 0.42;
    return h * (v.xy / v.z);
}
```

This ParallaxOffset function however is very simple if we peek inside UnityCG.cginc. This function simply changes the height and normalizes this towards the view direction. The problem with this simple parallax function is visible if we look at the shader from a different angle, a lot of artifacts can be seen(image on the right). The reason for this is because the camera is actually looking at a flat plane, the wrong vertexes are seen if the camera is at an angle.

To visualise the problem with this simple parallax map, I've drawn what causes the artifacts.
The reason this effect looks correct from the top, is that the camera actually sees the correct vertex (cam A). But if we look at it from an angle (cam B) we see vertex C since we're looking at a flat plane, instead of the vertex we should be seeing (vertex D). To solve this problem I've implemented Parallax Occlusion Mapping. This

problem is explained in the POM interpretation by Zink (2013).

Parallax Occlusion Map

First we need to transform the vertex vectors to tangent space. This is to make sure everything uses the same unit system (this is because we need height & normal information at each pixel point).

These transformations are done in a function (cginc file) which is called by the vert function of the shader to transform the proper coordinates.

```
void vert(inout appdata_full IN, out Input OUT) {
    //Setup transformations
    parallax_vert( IN.vertex, IN.normal, IN.tangent, OUT.eye, OUT.sampleRatio );
    OUT.texcoord = IN.texcoord;
}
```

The next part is to calculate the offset from the surface function.

To do this we first setup some default values, the maximum parallax offset, stepsize based on amount of samples and the change of surface geometry (ddx, ddy). Then to find the intersection of the eye vector with the heightmap (the place we're supposed to see, vertex D from the drawing above) we keep looping until it is found. Or until the current sample checked exceeds the amount of samples defined by the user. The less max samples the less steps are taken to find the intersection. When the height of the taken sample exceeds the height of the ray, it means the sample has gone past(below) the intersection. Then we take a point in between the last and current sample to approximate the intersection. Fewer samples will cause a less accurate approximation of the intersection and will cause imperfections and a less smooth surface.

```
while ( currSample < sampleAmount )
{
  currSampledHeight = tex2Dgrad(heightMap, texcoord + currentOffset, dx, dy ).r;
  //Compare eye vector with heightmap
  if ( currSampledHeight > currRayHeight )//Intersection found, set offset & break lo
  {
    float delta1 = currSampledHeight - currRayHeight;
    float delta2 = ( currRayHeight + stepSize ) - lastSampledHeight;

    float ratio = delta1/(delta1+delta2);

    //Intersection between current & last sample
    currentOffset = (ratio) * lastOffset + (1.0-ratio) * currentOffset;

    //Break out the loop
    currSample = sampleAmount + 1;
  }
  else//No Intersection found, keep looping
  {
    currSample++;

    currRayHeight -= stepSize;

    //Set current & last offset for the next loop
    lastOffset = currentOffset;
    currentOffset += stepSize * maxOffset;

    //Set last sampled height for next loop
    lastSampledHeight = currSampledHeight;
  }
}
//Return offset when intersection is found
return currentOffset;
```

It won't look perfect even with a lot of samples, but anti-aliasing should clean the rugged edges. When the intersection is found, the loop breaks and the offset is returned and the texture will be moved accordingly. This way the camera sees the surface of vertex D instead of vertex C (drawing above).

If there is no intersection found we lower the ray and move the offset to keep checking. This is how the POM looks:

To create POM I used a SIGGRAPH presentation by Tatarchuk (2006) and an interpretation by Zink (2013).

Emission Map

The lava still looks a bit dull. What this shader is missing is emission, this will give the light parts a lot more glow and make the solidified parts darker. To make use of the emission map, we need to add a shader feature to the #pragma:

```
#pragma shader_feature _EMISSION_MAP
#pragma target 3.0
```

By using this, light intensity will now have an effect on the shader based on the emission map.
In my case I used the same texture for the albedo and the emission map, this way the effect gives a nice glow. The next step is to get the emission texture and add it to the maintexture color. By multiplying this with an adjustable color we can change the hue of the lava to make it more colorful if needed.

We can also add some metallic and gloss to the shader to allow more adjustments:

After tweaking some values, we get vibrant/colorful lava:

Implementing a emission map was done using a tutorial by Flick (2019).

Distortion Effect

While it definitely looks warm, we want to make it look more dangerous and hot. We can make it look like its emitting heat by creating a distortion effect above the lava. To make use of Unity Post-processing stack V2, this default Unity Post Processing package can be added in the packages window. To use the Post-processing stack V2 I followed a tutorial on how to implement a custom distortion effect (Makin' Stuff Look Good, 2019)

We can't just distort the screen, since post processing effects are 2D it will just distort anything on screen with no relation to space. We do want to distort the screen, but only in certain areas

for a certain amount of blur. We could do this by implementing a particle system that creates particles that can be used for blurring, since particles are usually 2d anyways.

Since we want to create multiple blur effects(for each particle), we should create something to manage all these effects. We also want to be able to add these effects to the post processing stack.
DistortionManager:
Using a list of distortion effects, each distortion effect is added to the rendering pipeline using a command buffer. The command buffer allows extending the rendering pipeline.

```
//Register distortion effect
public void Register(DistortionEffect distortionEffect)
{
    _distortionEffects.Add(distortionEffect);
}

//Deregister distortion effect
public void Deregister(DistortionEffect distortionEffect)
{
    _distortionEffects.Remove(distortionEffect);
}

//Pass the added distortion effects to the commandbuffer to add it to the rendering pipeline
public void PopulateCommandBuffer(CommandBuffer commandBuffer)
{
    for(int i = 0, len = _distortionEffects.Count; i < len; i++)
    {
        //Get each effect in the list
        var effect = _distortionEffects[i];
        commandBuffer.DrawRenderer(effect.Renderer, effect.Material);
```

The DistortionEffect is a MonoBehaviour class that we can attach to the particle system to get the material and renderer and register them as distortion effects using the manager class above. We can also disable the actual (particle) renderer since we don't actually need to see the particle as we're just using it to distort the image.
Now that we have listed what needs to be a distortion effect, we need to create the actual effect. We want to cover the whole area with heat waves, so using the particle system we can cover the complete area with particles. Using a custom shader we can change the look of the particles. We'll also need some zCulling to prevent the shader from distorting objects in front of the particles.

First we need to sample the particles, which in my case is a normal map. I'll be using a smoke normal map. To use these normal map values, we have to unpack the normal map first. Then we can use the alpha of the texture to amplify the strength of the distortion.

To prevent the distortion effect from being drawn over objects in front of the lava we need to use zculling. To do custom zculling, we calculate the distance (eyedepth) to the particle in the vertex

shader. Then in the fragment shader we remap the sampled camera depth texture by decoding it to get the depth of the scene. Now we can compare the distance and check if there is something in front of the distortion particle. When the depth of the scene is greater than the depth to the particle (particle is in front of object in scene) the particle should be drawn.

Now that we have the “distortion” particle set up, we can use this to create the actual distortion itself. We do this by using the normal map particle that was set up earlier and using the (xy)texelsize of the screen to magnify the change of color(based on the moving normal map).

Now we need to apply this to each particle that was added to the distortion list in DistortionManager and render this to the screen. We call the distortionmanager to add the effects to the command buffer. We finally blit the source to the destination in the command buffer using the distortion shader (sheet) to transform the screen to the screen with distortions.

Flowmap

To implement a flowmap I used a tutorial by Flick (2019).
To create directional movement we create a function to move the uv over time. We can use this function to set our own direction and pass the rotation matrix to make sure the Parallax Occlusion Map is corrected if the uv has rotated.

We then use this function to create a grid with cells that each have their own directional flow. Here we sample the flowmap to use as the flowvector to make sure each cell moves in the correct direction.

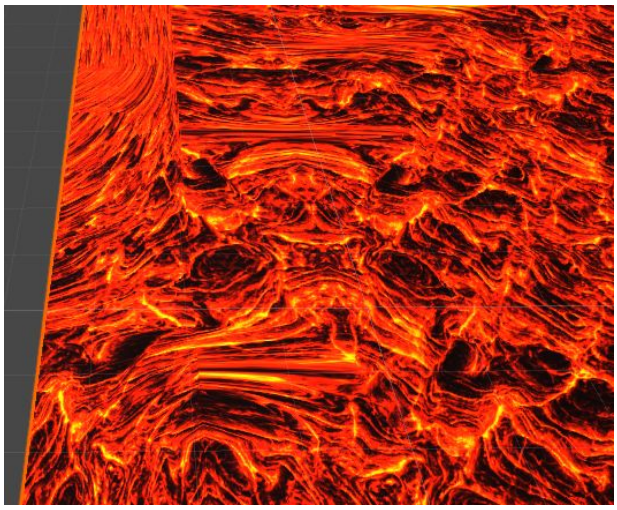
```
float2 FlowCell (float2 uv, float2 cellOffset, float time, out float2x2 parallaxRota
    float2 shift = 1 - cellOffset;
    shift *= 0.5;
    cellOffset *= 0.5;
    float2 uvTiled = (floor(uv * _GridResolution + cellOffset) + shift) / _GridResol

    //Tile based flow
    float3 flow = tex2D(_FlowMap, uvTiled).rgb;
    flow.xy = flow.xy * 2 - 1;
    flow.z *= _FlowStrength;
```

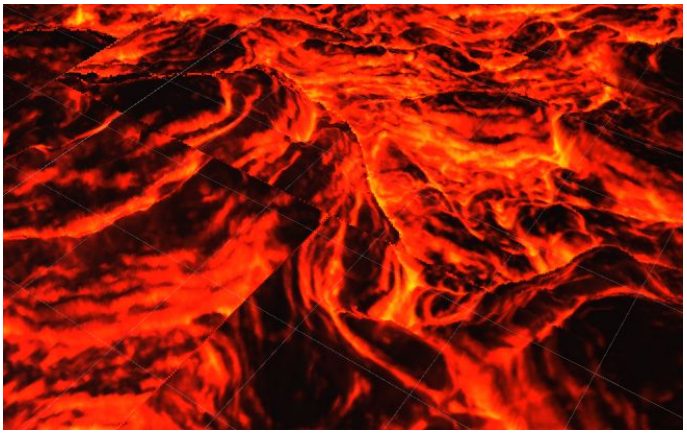
```
float2 UVFlowA = FlowCell(uv, float2(0, 0), time, parallaxRotation);
float2 UVFlowB = FlowCell(uv, float2(1, 0), time, parallaxRotation);
float2 UVFlowC = FlowCell(uv, float2(0, 1), time, parallaxRotation);
float2 UVFlowD = FlowCell(uv, float2(1, 1), time, parallaxRotation);
```

The next step is to use the cell offset to make sure the change in direction is aligned. This is where problems occurred, aligning the texture(using celloffset) caused stretching in extreme changes in direction. On the right I set different offsets for the tiled uvs. Without aligning the texture it is very obvious where the texture is cut off to flow into a different direction. This stopped me from completing this feature; see the images below for flow with and without alignment:

With alignment of flow:



Without alignment of flow:



There were no examples of someone implementing a flowmap with POM, the only thing i had as inspiration was a forum post by mouurusai (2019).
As a temporary solution, I added a slider to the shader where the direction can be set manually. I also added clockwise and counter-clockwise rotations that can be turned on or off.

Tiling

To allow for different sizes of meshes & planes, I added a way to tile the texture in both the x and y direction. I also added the offset. I thought this feature was finished and didn't look at it too much, later I found out that rotating the plane/mesh breaks the tiling. It was too late to find a fix for this, so I commented it
out.

When the mesh/plane is not
rotated (0,0,0) it works correctly so use at own risk.

As a final scene I used a demo scene from: the Top-down Asset pack by Manufactura K4 5(2019).
The final result looks like this:

Reference List.

Alisavakis, H. (2019, February 21). My take on shaders: Parallax effect (Part I) – Harry Alisavakis. Retrieved from https://halisavakis.com/my-take-on-shaders-parallax-effect-part-i/

Flick, J. (2016, October 31). Rendering 9. Retrieved from https://catlikecoding.com/unity/tutorials/rendering/part-9/

Flick, J. (2018, June 29). Directional Flow. Retrieved from https://catlikecoding.com/unity/tutorials/flow/directional-flow/

Makin' Stuff Look Good. (2019, April 1). *Shaders Case Study - Distortion FX with Unity's Post-processing Stack v2* [Video file]. Retrieved from https://www.youtube.com/watch?v=xH5uUfeB2Go

mouurusai. (2019, November 12). flow map + parallax map [Forum Post]. Retrieved from https://forum.unity.com/threads/flow-map-parallax-map.776573/

n00body. (2015, January 2). [SOLVED] Detail mapping + Parallax = Texture Swimming? [Forum Post]. Retrieved from https://www.gamedev.net/forums/topic/664276-solved-detail-mapping-parallax-texture-swimming/

Tatarchuk, N. (2006, March 14). Practical Parallax Occlusion Mapping For Highly Detailed Surface Rendering [Slides]. Retrieved from https://developer.amd.com/wordpress/media/2012/10/Tatarchuk-POM.pdf

Zink, J. (2013, August 9). A Closer Look At Parallax Occlusion Mapping. Retrieved from https://www.gamedev.net/tutorials/_/technical/graphics-programming-and-theory/a-closer-look-at-parallax-occlusion-mapping-r3262/

Manufactura K4 5. (2013, June 13). *Top-Down Caves*. Retrieved from https://assetstore.unity.com/packages/3d/environments/dungeons/top-down-caves-39124