CMPM 163 Final Project
Extended Abstract

Kevin Teng Wu
University of California, Santa Cruz
California
ketwu@ucsc.edu

Shuo-En Li
University of California, Santa Cruz
California
sli112@ucsc.edu

Kevin Wu
University of California, Santa Cruz
California
kwu28@ucsc.edu

Joey Sandmeyer
University of California, Santa Cruz
California
jsandmey@ucsc.edu

*Not an error, literally two Kevin Wus in the same course

ABSTRACT
The CMPM163 Final Project consists of several shaders with different effects. All of the shaders are used to create research lab environment. Each team member is responsible for a single shader. Kevin Teng Wu made a melting shader that mimics the phenomenon of melting by using vertex displacement and noise to achieve such an effect. Kevin Wu made a parallax shader that gives a more convincing illusion of depth than other bump mapping techniques. The parallax shader uses a displacement map to determine how deep each part of the mesh should appear without using triangles to do so. Shuo-En Li made shaders that bring about a jelly effect by implementing translucency, transparency, and vertex displacement. Joseph T Sandmeyer created a render to texture and crt effect.

CCS CONCEPTS
GLSL, vertex, fragment

KEYWORDS
Melting, vertex displacement, noise, translucency, transparency, ray marching

ACM Reference format:
Diary.conewars.com
learnopengl.com/Advanced-Lighting/Parallax-Mapping
www.habrador.com/tutorials/shaders/3-parallax-mapping/

https://www.slideshare.net/colinbb/colin-barrebrisebois-gdc-2011-approximating-translucency-for-a-fast-cheap-and-convincing-subsurfacescattering-look-7170855
http://farfarer.com/blog/2012/09/11/translucent-shader-unity3d/
https://unity3d.com/learn/tutorials/topics/graphics/making-transparent-shader

1 INTRODUCTION

1.1 Melting:

Melting is a natural phenomena in which a solid changes into a liquid at a fixed temperature, or melting point. It requires a certain amount of heat in order to occur. The reverse of this is called freezing, when a liquid becomes solid. In terms of physics and thermodynamics, a liquid has more energy and instability than a solid, and it occupies a larger volume than a solid.

If we think about real life through 3d objects, a solid is a static, unchanging mesh that is highly subdivided. At a certain temperature, or the melting point, the mesh slowly starts to "melt." The vertices of the object slowly flow down and outwards. But the amount of vertices that flow away from object will ultimately depend on the viscosity, or flow, of the substance.

In terms of GLSL shaders, most of the calculations will be done in the vertex shader. We will have to focus careful on vertex displacement and provide external variables to the shaders, like time. Just like how ice cream melts, melting is rarely uniform; the whole mesh does not become liquid at the same time. Thus, a noise texture or value will be used to drive the shape of the liquid puddle.

1.2 Jelly:

The attempt was to create a jelly-like creature that simulates the visual effect of Jelly Fresh, a game character in Splatoon. To achieve so, there are some key features in Jelly Fresh's appearance that shall be recreated. The **inner body part**, which is translucent, and the **outer body part**, which is transparent and see-throughable, as well as its jiggly, bloaty **movement,** which seems to be an unavoidable quality shared among all Jelly species. From now on, let's call this replica of Jelly Fresh-- Blob.

Before we dive into this further, here's some information about translucent shader. Translucency is the "quality of allowing lights to pass through a media diffusely and partially" (Colin Barre-Brisebois - GDC 2011 - Approximating Translucency …). Translucent shader, despite not physically realistic, simulate the lighting effect on objects that the light can partially passed through. Its application in computer graphic is broad, ranging from the leafs under sunlight, jade texturing, to weird creatures like Blob. With this shader, some interesting visual effects can be created through interaction with different light sources.

1.3 **Parallax:**

Bump mapping is a technique in computer graphics that simulates the appearance of bumps and wrinkles on a 3D mesh without actually using any polygons to do so. Typically a grayscale texture or a normal map is used to determine which parts of the mesh should appear bumpy. The motive behind developing more convincing bump mapping techniques lies in the fact that first, the more triangles there are in a 3D mesh the more computational power that is needed to render that mesh and that second, many other bump mapping techniques do not provide a convincing enough sense of depth. Bump mapping is especially used in video games where optimizing graphics performance is crucial to achieving real time graphics because it lowers the number of triangles needed to be drawn on screen at any given time.

Parallax mapping is an enhancement of other bump mapping techniques. It gives a more convincing appearance of depth than other bump mapping techniques as shown below in Figure 1.

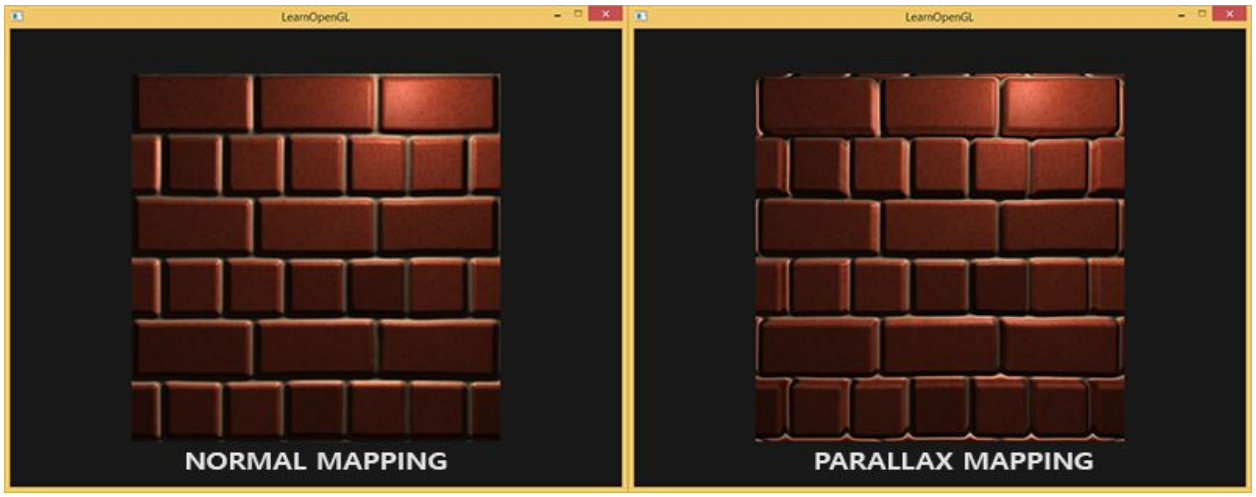


Figure 1 Normal Mapping VS Parallax Mapping

Parallax mapping saves on polygon count. The same brick wall above only has 2 triangles and 6 vertices. The wall would have had to have around 1000 or so vertices in order to achieve the same result without Parallax mapping.

1.4 **Render Texture:**

The Unity Editor includes [Render Target Texture] as a field of Camera objects. Render targets are supported by modern GPU's as an alternative to rendering an image to the screen buffer or some off-screen buffer. The image might still show up on the screen buffer, but indirectly through a texture. Therefore, the scene has been rendered twice before reaching the user's irises. This technique can be bootstrapped; by aiming a render texture at itself, or by aiming two render textures at each other, a virtual hallway appears, spanning off into infinity.

For the lab scene, we integrated render textures as monitors for analyzing subjects. Some other render textures are placed around the scene for aesthetic purposes.

2 EXPERIMENTAL AND COMPUTATIONAL DETAILS

2.1 Melting Shader in Practice

Melting is a downward event because of gravity. As such, the vertex displacement is encoded as:

```
worldSpacePosition.xyz += float3(0, -1, 0) * rate * (_Time * 0.1);
```

As the mesh “melts,” which is to say that the vertices slowly move downwards, the vertices should also spread out and away from the object center. If we assume that 0 is not melted and 1 is fully melted, then the new vertex position is based on a percentage of melting:

```
//melt is percent at which vertex position has descended relative to original position
float melt = ( worldSpacePosition.y - _SurfPoint ) / _MeltForm;

//0 is not melted, 1 is fully melted
melt = 1 - saturate( melt );

//sample noise texture in red channel
float noiseValue = tex2Dlod (_NoiseTex, float4(v.texcoord.xy, 0, 0)).r;

//smoother step, looks like smooth bezier curve
melt = (melt*melt*melt * (melt * (6.0 * melt - 15.0) + 10.0)) ;

//vertices spread outwards
worldSpacePosition.z += worldSpaceNormal.z  * melt *noiseValue;
worldSpacePosition.x += worldSpaceNormal.x  * melt ;
```

The noiseValue variable is a noise value that can be from a sampled noise texture channel (such as the red channel) or a randomly generated value. The end effect is a slow meltdown of the object and its vertices spread outwards the more it melts. It is worth noting that diary.conewars.com has starting shader code that does melting.

2.2 Jelly Shader in Practice

(a) Translucency

Let’s briefly study the concepts behind translucent shader. This technique for computing translucency extracts from surface scattering, a mechanism that keeps track of how the lights scatter after penetrating the surface, in which the lights will traverse in different directions and angles based on the material.

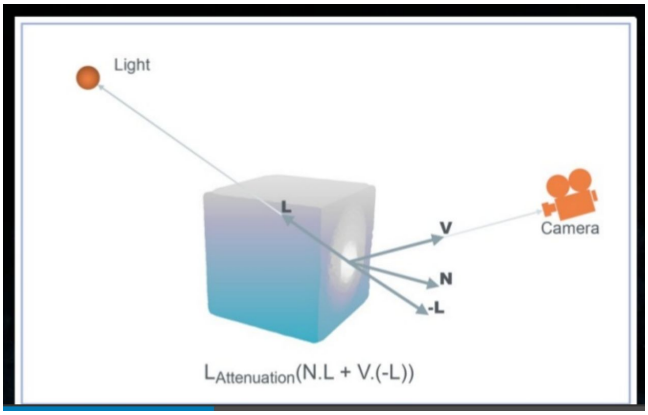


Figure 2: The concept behind translucency

Pseudocode:

```
transLightDir = lightDir + Normal * Distortion;
transDot = pow ( max (0, dot ( viewDir, -transLightDir ) ), Power ) * Scale;
transLight = (attenuation * 2) * ( transDot ) * Alpha * SubColor;
transAlbedo =  Albedo * LightColor * transLight;
```

First, we calculate the translucency light direction by adding the light direction with the production of normal vector and distortion. The value of distortion is calculated based on the shape and thickness of the texture on the object. Then, we calculate the transDot, which is like a specular, but instead of being on the surface, it's going to simulate an inward effect inside the object. Power and scale dictate intensity and range of this dot effect. Next on, we calculate the transLight. Attenuation calculates the fading off for the effect range of the light, to give the simulation a more organic result. The light also weakens when the alpha is lower. Lastly, we add the albedo(which is the texture color plus object color) with the light color and translight color. Note that the value obtained is only for the translucency, not the final color value to be outputted by the fragment shader. Because translucency is only for calculating how the light passed through the object affect the surface, we need another light calculation for the surface's reactant to lights. In this shader, we simply uses a phong lighting model to compute the outer light. (phong lighting's code is provided by Unity.) Eventually, we add these two results together to create the final value to be outputted by the fragment shader.

(b)Transparency

In Unity, transparent effect is very easy to obtain. The trick is done in these four lines of code:

- Tags {"Queue"="Transparent" "RenderType"="Transparent" }
- ZWrite Off
- Blend SrcAlpha OneMinusSrcAlpha
- Color.a = (Some value that is considerably less than 1, but more than 0)

(c) Vertex Displacement

Vertex displacement is achieved through adding displacement value that is produced by a variant of sin(Time) and cos(Time) functions to the x and y vertices respectively.

2.3 **Parallax in Practice**

Parallax mapping creates the illusion of displacement based on the information stored inside a texture. This texture is in the form of a grayscale height map where lighter areas make the mesh appear to bulge out and darker areas make the mesh appear to have an indent. The image below in Figure 3 shows the height map for the Parallax mapped brick wall from Figure 1 from section 1.3.

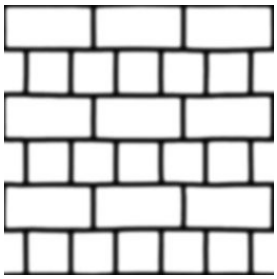


Figure 3: Height map that determines the bumpiness of the mesh surface

So how does Parallax mapping create the illusion of depth? It does so by firing rays from the view camera in tangent space towards a geometry. The rays are fired at a constant step size and we check during each step whether the ray is below that geometry. This is essentially raymarching. When the ray is below the geometry we obtain the weighted average of the last point on the ray. Where this last point will be will depend on how many steps the ray takes. Below in figure 4 is an illustration of what is going on.

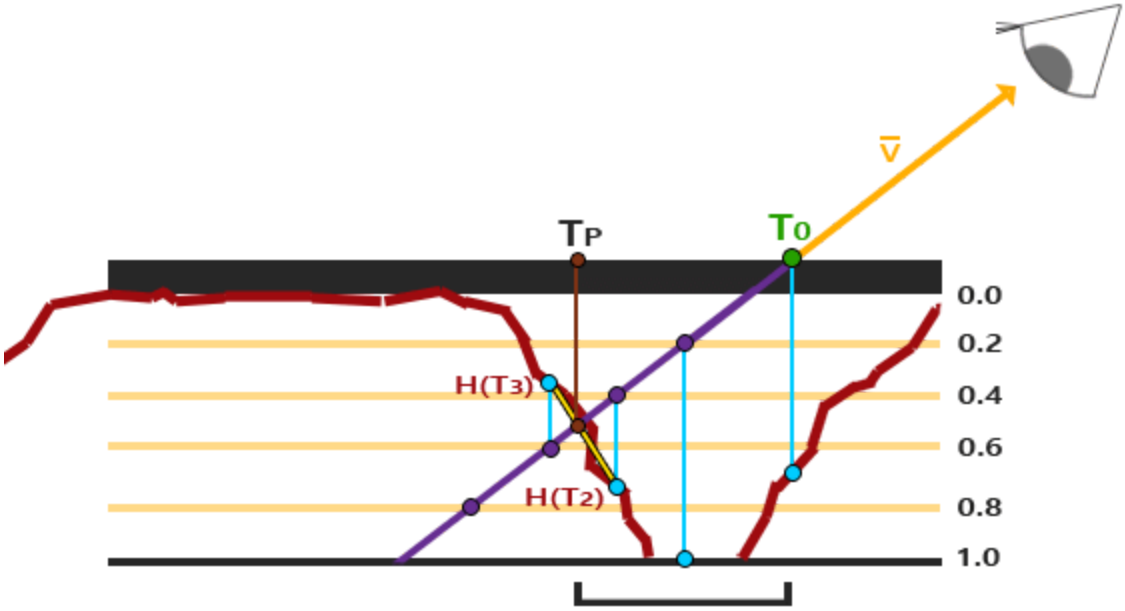


Figure 4: Illustration of the raymarching

Below is Pseudo Code of the Parallax shader:

The surface shader is where most of the work is going on:

surf(Input IN, inout SurfaceOutputStandard o)

In the body of this function is a for loop that raymarches stepAmount of times:

for (int i = 0; i < stepAmount; i++)

For every step we check whether the ray is below the geometry:

if (rayPos.y < height)

This compares the height of the geometry to the height of a point on the ray. If the ray point is above the geometry then we keep raymarching, however once the ray goes below the geometry we then find the intersection point of the ray with the geometry represented by the height map. This is done by obtaining a weighted average between the current ray position and the position of the ray's previous step.

Here is the function that does the weighted average:

```
Float2 getTexPos(rayPos, rayDir, stepDistance) {
        Return the prevPos.xz * weight + rayPos.xz * (1 - weight);
        Where weight = currentDistToGeometry / (currentDistToGeometry -
prevDistToGeometry);
}
```

It takes in the ray position, direction and the step distance which is used to get the previous step position. Finally we break from the for loop.

2.4 **Render Texture in Practice**

Render targets are used as the backend function for shaders that need to store data in multiple bitmaps at once. For example, deferred shading requires, for every pixel the buffer, information about the pixel's color, its normals, its depth, etc. This data can be stored in multiple render targets.

Render textures can mimic the function of a mirror, though the image in the mirror will not have any apparent depth, since it won't parallax when the user strafes. Perhaps an implementation using Kevin's parallax shader could remedy this problem? To achieve such an effect, the depth of the pixels from the camera plane would have to be recorded.

3 RESULTS AND DISCUSSION

3.1 Melting in Action

The scene shows two variations of melting, based on how much the vertex shader and supporting c# script are responsible for the effect. In the first example, the shader and script share a fair amount of responsibility. In the second example, the shader assumes most of the responsibility for creating the effect.

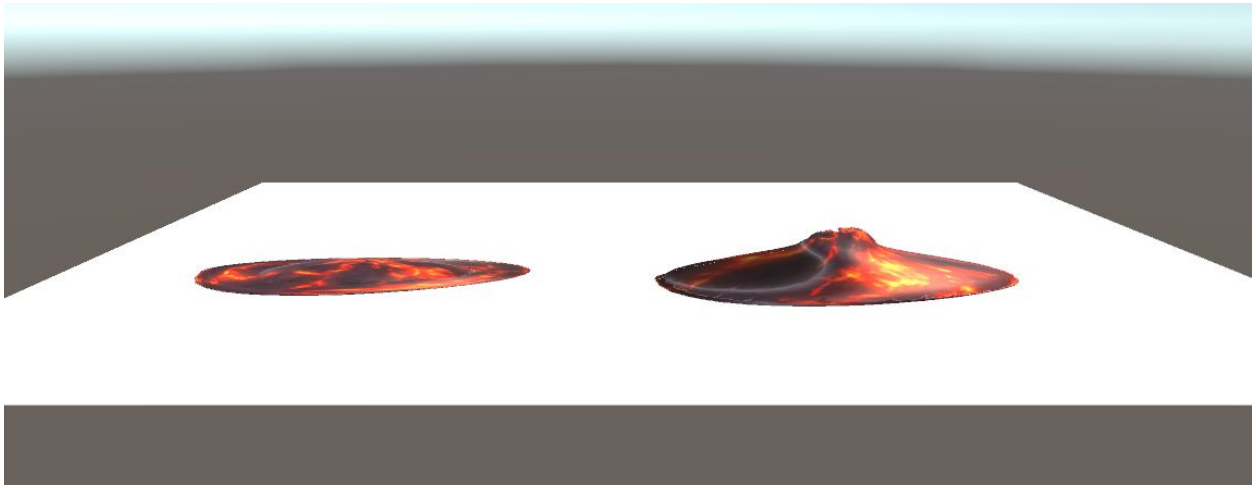


Figure 5: First puddle on the left with shared responsibility. Second puddle on the right with mostly shader responsibility.

Both shaders use the melt percentage formula:

```
melt = (melt*melt*melt * (melt * (6.0 * melt - 15.0) + 10.0)) ;
```

We will first be going over the specifics of the first example. This shader passes off some of the melting behavior to a C# support script. At the start of the simulation, a Unity Coroutine is executed, which is like asynchronous instructions in a program. The transform of the object position moves downward towards a set ground position.

Psuedo code for c# script:

```
Vector3 startPos = transform.position;
Vector3 endPos = Vector3 (endPosition);

float startTime = 0.0f;

while (startTime < duration) {
        timeAtPos += Time.deltaTime;
```

```
            float percent = Mathf.Clamp01 (timeAtPos / duration);
            transform.position =  lerp (startPos, endPospercent);
            //lerp shader.meltVar;
            //lerp shader.meltForm;
    }
```

There are extra shader uniforms that are lerped (or tweened) to enable the melting. MeltVar affects the spread of the puddle and meltForm affects the shape of the top pudding like shape of the mesh. A noise texture is sampled as specified in 2.1 Melting in Practice and used to drive the shape of the puddle. Together, the c# script code and shader code from 2.1 give the object a nice and smooth melt down onto the plane.

In the second example, we only use the script to pass the changing uniform meltVar. The uniform meltVar, again, affects the size of the puddle. The shader assumes most of the functionality of the melting and uses the vertex shader to create a similar effect as seen in the first example. All object transform changes are in the vertex shader as downward vertex displacement and spreads outwards according to shader code from 2.1.

```
worldSpacePosition.xyz += float3(0, -1, 0) * rate * abVal * (_Time * 0.1);
```

The effect is rougher but simpler. We do not rely on the C# script to drive our downward melting behaviour.

3.2 **Jelly Observations**

Playing around with different light sources, I noticed that the translucent shader indeed creates an interesting color spectrum on Blob's body. The light spots are especially illuminated. However, the feel of translucency is not so prominent. I assume it's because Blob's body is a sole geometric sphere without a texture map applied to it. Thus, the translucent light paths are uniform and the alpha isn't varied. Moreover, adding the transparent and textured outer body has obscure the visibility of the inner body more or less.

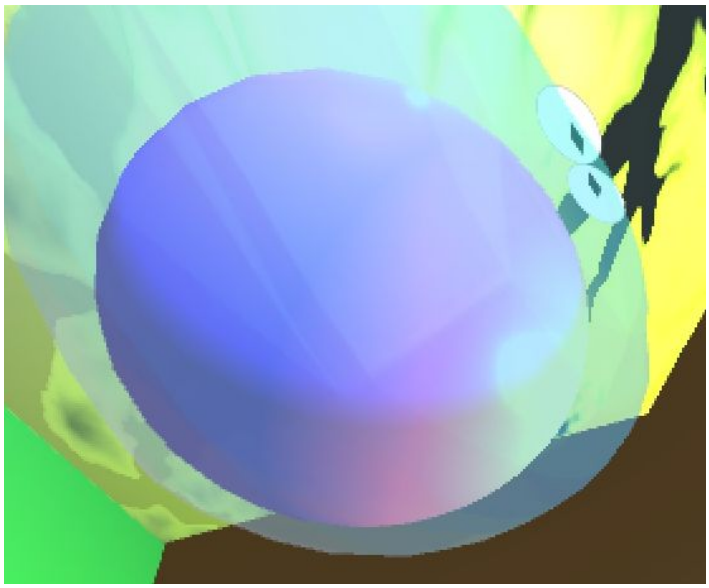


Figure 6 Blob the Beauty

From my observation, the ideal value for surface Power seems to fall between 0.7 and 1.5, if the intention is to create a more obvious translucency effect. The surface Scale value should be at least 20, and for visible change in effect, scale it up in increment of 10. It is important to choose a light color that contrasts with the object's color, so the falloff of the specular and color gradient is more obvious.

As for the transparent outer body, the alpha value set at the range between 0.4 and 0.7 creates a nice balance between showing the texture color and maintaining transparency. With a higher value, the texture color starts to take over transparency, and with a lower value, the transparency starts to drive toward the edge of invisibility.

The values for the vertex displacement was hard to decide. It can't be too simple so that the movement feels bland and too predictable, but it also can't be too intense or too arbitrary so that Blob looks like it's crazy. After ample experimentations, the equations are finalized as the following:

Inner body:

```
v.vertex.y += sin(_Time.y * 4 + v.vertex.x *8) * 0.04 + cos(_Time.y * 4 + v.vertex.x * 8) * 0.05  ;
v.vertex.x += v.normal * clamp(sin(_Time.y) * 0.05, 0, 2);
```

Outer body:

```
v.vertex.x += clamp(sin(_Time.y) * 0.01, 0, 1);
v.vertex.y += clamp( 0.03 * sin(_Time.y * 4 + v.vertex.x *8)  - cos(_Time.y * 4 + v.vertex.x * 8) * 0.05, 0, 1)  ;
```

Plus a C# script that scales Blob up and down through a sine function of time, these codes create wavy movement for Blob that are subtle but interesting.

3.3 **Parallax mapping in action**

Below in Figure 7 is parallax mapping for a floor.

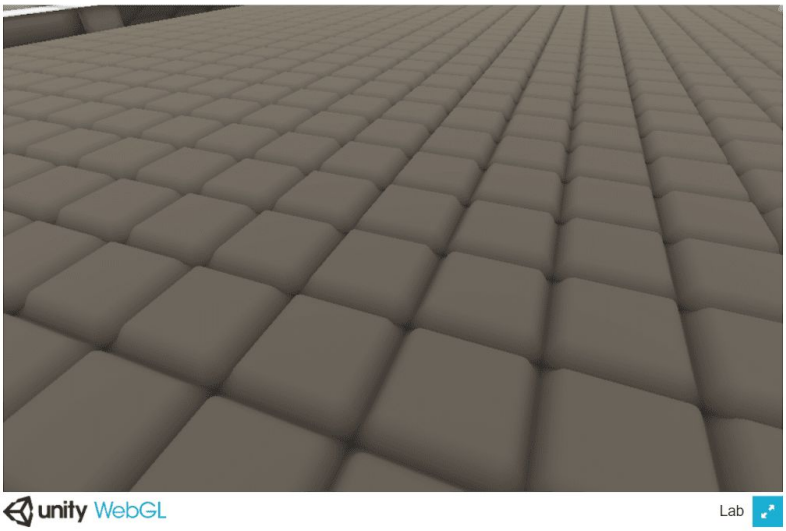


Figure 7: Parallax mapped floor

Here is the floor's corresponding height map.

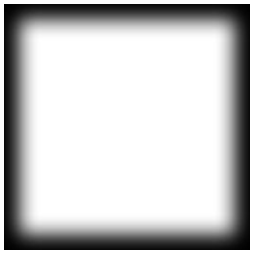
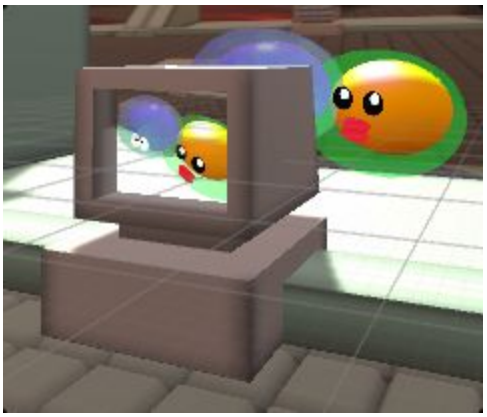


Figure 8: Tileable heightmap

I exposed a height value that went from 0 to 5 to control the strength of the displacement. 0 meant no displacement and 5 meant the strongest displacement.
In my observations I noticed that subtle changes in the height value dramatically affect the depth of the tiles. I found a value of 1 to 2 to create a deep enough floor.

3.4 **Render Target Experimentation, Implementation, and Post-Processing Effects**

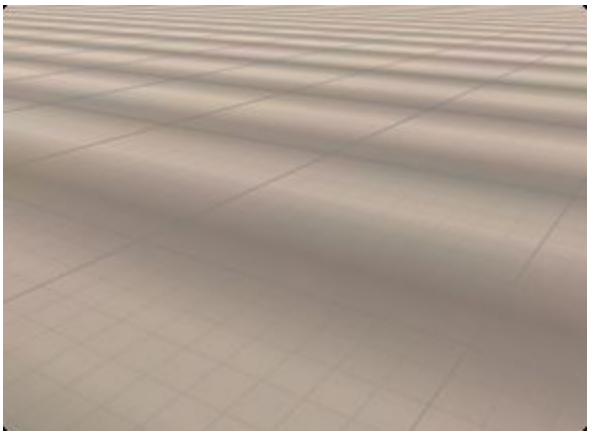
There are some wonderful video games that have integrated real-time render textures as an essential part of the gameplay experiences. Five Nights at Freddy's, a recent horror hit, gives the player access to security monitors that show the player locations where monsters are hiding. The player participates in mapping the disjointed space from the monitors into a 3D space, and by understanding the consequences of 3D space, the player can better protect their avatar's wellbeing.



My role in our Lab scene involved implementing render target monitors to showcase the work my peers have done on this project.

I began with the standard render texture pointing at each subject. There's a sort of mystique here that I love: seeing a virtual computer monitor displayed on my computer monitor is really rocking my metaphysics.

I layered another plane over the monitors with a shader that emulated scan lines. Unfortunately, this shader was incompatible with transparency, and it always renders on top of the other geometry in the scene. My solution was to capture this effect in a bitmap texture, and apply it as a transparent layer over the render textures.



It all worked out very well. I got the desired effect, where scan lines appear to form curves as a result of misaligning with the camera's pixel array. It is the same effect that is achieved when a video camera records a tv or computer monitor. The monitors stand out, since the scan lines shift widely when the user moves their view.

I experimented further with Unity's standard shaders. The specular shaders look especially pretty when applied to render textures. I settled on a chroma shift effect that made Shuo-En's Jellyfish really pop. And Kevin likes the way the walls look with render textures, so we are using them for the walls as well.

4 CONCLUSIONS

4.1 **Melting Conclusion:**

An effective melting shader will use both the shader and c# script to create a believable melting effect. With adequate tweaking of the uniforms, the object will truly look like melting ice cream that shrinks in size and spreads all over the place. It should also be noted that the melt percentage formula can be changed to any other function to output a percentage, such as an exponential curve.

However, there are a some drawbacks to this shader. One, the melting shader is not physically accurate. This means that, if the object is on a raised platform, the puddle will still spread out as if it is on flat ground. This is because the vertices are being displaced on the GPU, while physic calculations are done in the CPU. There is no additional logic to compute these effects. Two, you have to rely on an external script to some extent to change the uniforms over time. Without an external script, the melting object is pressed to spread out in an adequate manner.

It should be noted that, during the creation of the melting shader, tessellation was employed to make the melting object smoother and more liquid-like. But the built scene in WebGL, which runs in a lower target value, was unable to run the shader as tessellation is employed in Unity's pragma target 4.6. So there is no tessellation involved in the final creation as presented. But you can enable it in code by un-commenting out the #pragma target 4.6 in the shader code and the tessellate:tessFixed.

4.2 **Jelly Conclusion:**

There are some notable technical challenges. One is the hardship to create an outline highlighter for Blob. I was hoping to add outline highlights for Blob inside the transparent shader, but failed to do so. A semi outliner shader that can't be used in conjunction with transparency is created as the byproduct of such failure. This shader is implemented on the other Jelly creature in the scene. I would also love to have a transparent shader where the transparency changes according to the vertex's distance from the center, so there can be a gradient effect on the transparency. Disappointedly, I've only worked up one where the alpha changes based on time, not position. This unsatisfactory byproduct is also implemented on the other Jelly creature.
Originally, my ambition was to create a shader that combines toon effect with translucency. After gaining more understanding on the subject as well as some experiments , I realize these two are really difficult to combine, because toon shader generalizes the light colors and groups them into sections, while translucent shader can't maintain its effect if generalized because then the albedo and colors won't be accurate. As a result, I made a separated toon shader for the other Jelly creature. Basically, this other Jelly creature is the aggregation of failures.
Overall, the visual result on Blob is quite satisfactory. It has a decent look with a carefully decided movement that emulates the jiggling. However, in the actual scene, colorful light sources are lacking and thus Blob's appearance is not the best it can be. The three key features in jelly-- translucency, transparency, and vertex displacement-- are definitely interesting and widely-applicable effects that I'll continue working on in the future.

4.3 **Parallax Conclusion:**

For challenges, understanding the math behind the technique took a long time to grasp. But getting the effect to finally work felt really rewarding. In terms of how the shader went, overall parallax mapping has a more convincing bump than other bump mapping techniques. However, there are some issues. I noticed that making a very steep displacement causes the texture applied to the mesh appear stretched. Also for the parallax effect to work at very far distance, a high step amount had to be set. In other words to get more accurate parallax results one has raymarch across more steps. Raymarching too few steps results in ugly artifacts when viewing the mesh.

4.3 **Render Texture Conclusion:**

The Unity Engine is powerful. Implementing render textures only takes a couple of clicks using Unity's standard Camera module. The effect might be easy, but making it look good can be difficult. The render textures don't react to shaders and lighting like a typical bitmap texture does, in part because they are rendered in real-time. Without at least a little bit of post-processing fiddling, the render texture will not appear to emit light like a modern backlit graphical display does, and it is missing other qualities that the human eye can easily spot in real-life electronics. Luckily, a shader can be fashioned to give render textures any effect that is desired. They can appear distorted, amplified, partially transparent; anything. The effect usually requires a different technique than one would use on a standard material, so the shader programmer's mind has to be reoriented to thinking about realtime pixels.

References:

https://forum.unity.com/threads/need-some-help-writing-a-simple-scanlines-shader.375868/

https://www.youtube.com/watch?v=pA7ZC8owaeo

ACKNOWLEDGMENTS
This work was created in the CMPM 163 course under the course instructor Angus Forbes.