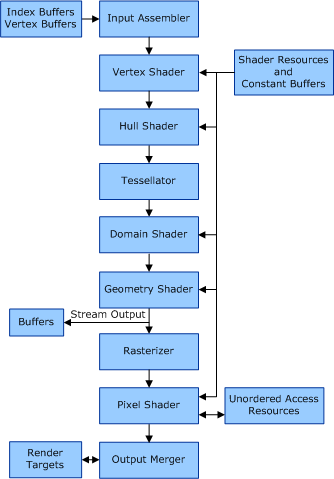
The Rendering Pipeline

As shown in the diagram, the DirectX Rendering Pipeline consists of a variety of stages. While only a few are programmable, (like the pixel shader, the geometry shader, and the vertex shader), all are important when generating real-time graphics.

The first stage of the pipeline is the Input Assembler. In the assembler, data from the buffers is read and assembled into primitives that that will be used by the following stages. All this data is then output into the vertex shader.

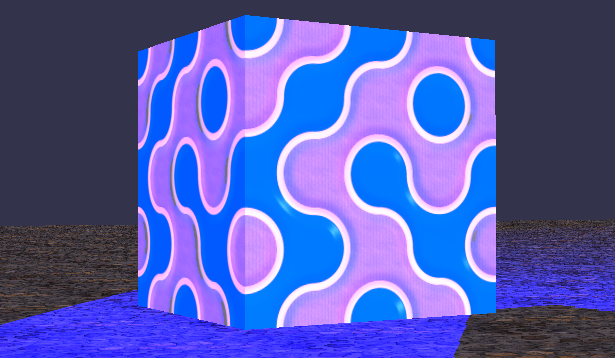
According to Microsoft contributors Kennedy and Satran (2018), the first shader, (the vertex shader) “processes vertices from the input assembler” and usually focuses on transformations. For the pipeline to execute, it is imperative that the vertex shader is active, even if no handled modifications are required.

The pixel shader takes vertex values from the rasterizer stage and produces per-pixel colour values. When models in the project are affected by texture depth, for example in the normal mapping cube, the pixel shader can also output depth values, however this is not necessary to achieve such effects – for the parallax mapped cube, the depth is changed in the ‘Scene’ file.

The last relevant stage of the pipeline is the output merger, which generates the final scene via a combination of all utilised shaders and all buffers, “The OM stage is the final step for determining which pixels are visible (with depth-stencil testing) and blending the final pixel colours.”

Specular Lighting

Specular lighting involves the use of a texture of the alpha layer of a texture to replicate a reflection. Since the surface of the teapot is smooth, the highlight is focused and a clear reflection of the light source is produced, (circled above). This is achieved via a series of calculations in the teapot’s pixel shader.

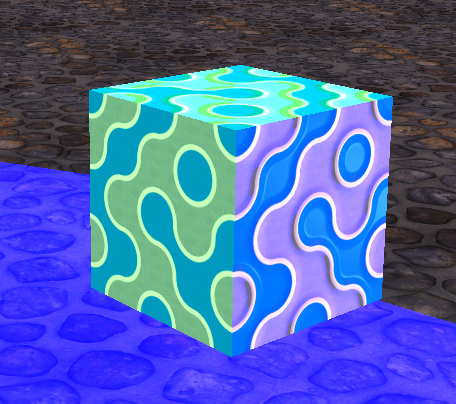
Normal Mapping

The process of normal mapping uses a texture map that contains normals. Every normal in the texture is read and applied to the cube model. The bumpiness is further emphasised when the cube is affected by the surrounding spotlights. Normal mapping does not change the structure of a model in any way –bumpiness is only implied.

Parallax Mapping

Parallax mapping is very similar to normal mapping in that it is applied to models in the same way. However, parallax mapping attempts to show depth as well as bumps. This effect is also ‘fake’ in that it only implies model distortion.

Depth is emphasised via the use of a height map stored in the alpha channel.

Directional Lighting

//Light 3

float3 colour = {0.0f, 0.0f, 0.0f };

float3 lightDirection = -gLight3Direction;

float lightIntensity = saturate(dot(modelNormal, lightDirection));

if (lightIntensity > 0.0f)

{

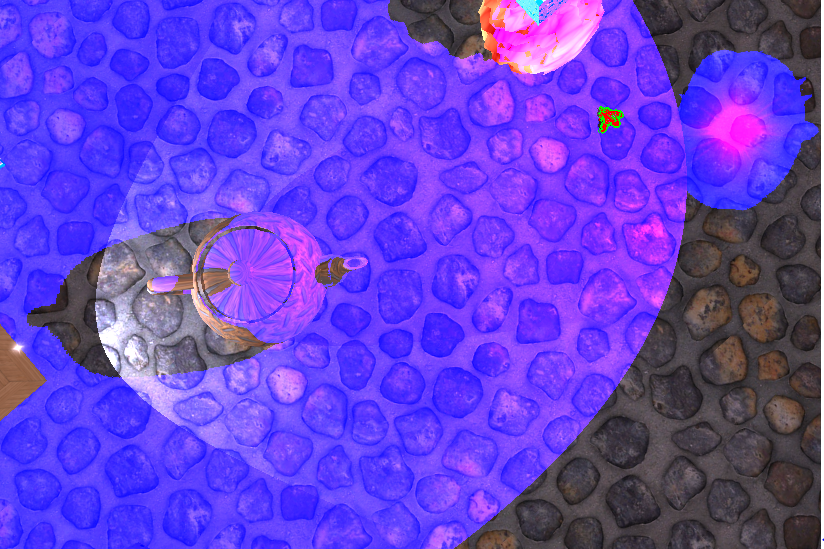
colour = saturate(gLight3Colour \* lightIntensity)

}

In the program, the directional light only affects the cubes in the scene, and hits them at an angle that affects the top and the left facing side of the cube only. As with the nature of directional lighting, neither display attenuation.

This is implemented into code through the inclusion of an extra light in each respective pixel shader. The colour produced by the above code is added to the final colour value and produces the above result.

Spotlight

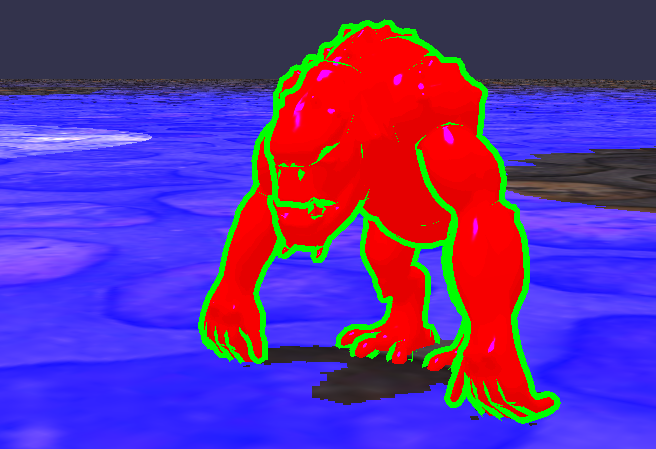
Spotlights are lights restricted to a conical shape that only point in a chosen direction. Light attenuates the further away from the origin point affected pixels are.

They use point, direction and colour in their equations. The following code describes the white light that circles the teapot model.

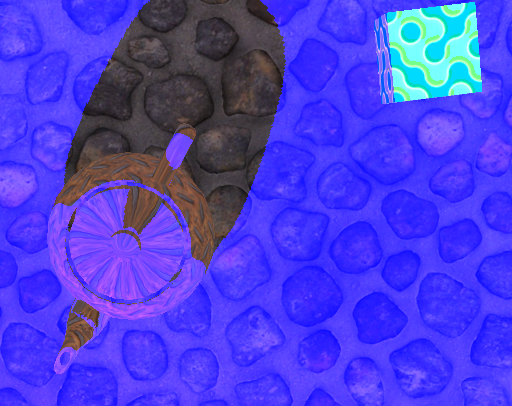
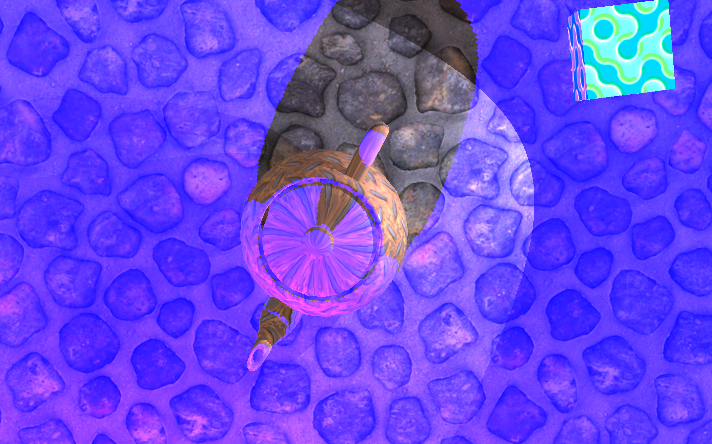


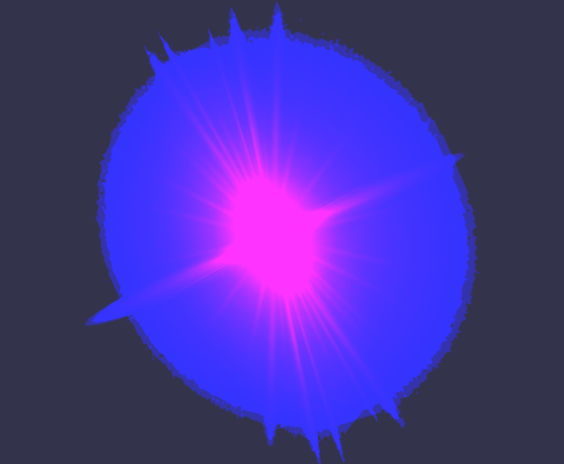
Shadows

Shown in nearly every example picture, nearly every model used projects a shadow. This is due to the scene being rendered from the point of view of both spotlights. Pixels are tested every frame against the shadow map to see if it is in direct line of a light source. If not, the pixel appears shadowed. The depth is adjusted so polygons do not shadow themselves. Ambient light and directional light do not produce shadows in the project.

Cel Shading

Though cel-shading can be done in multiple different ways, the above result was produced by rendering two versions of the troll model, one of which was turned ‘inside-out’ and set to a bright green. The colour of the ‘main’ troll model is defined by a 1D-texture that is affected by surrounding lights, emphasised by the pink tints and the darker shaders of red.

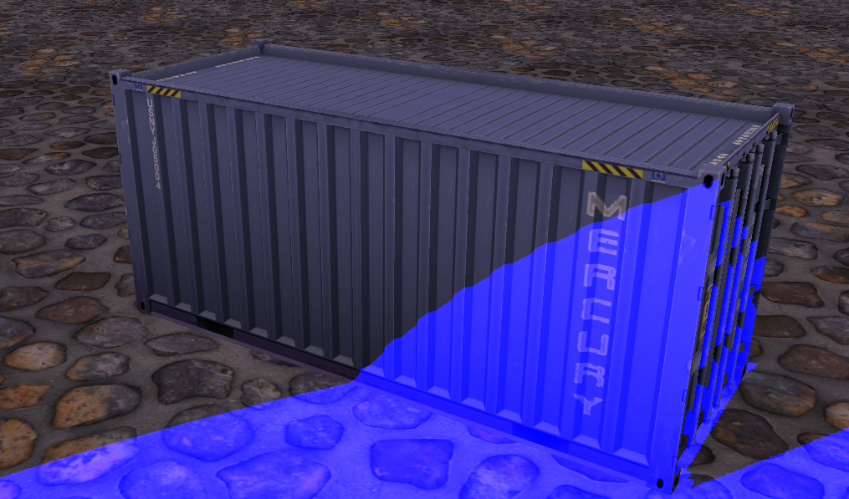
Light One changing colour and Light Two flashing on and off



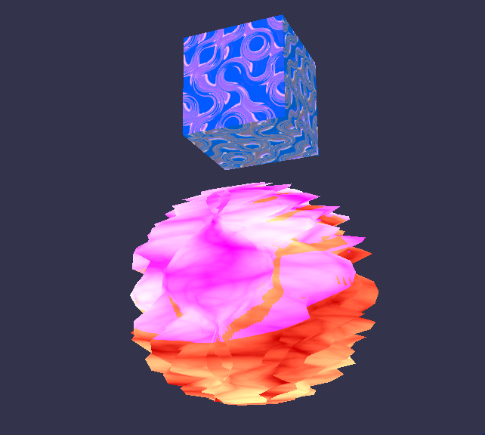
Attenuation

Both spotlights are affected by attenuation, in which the original colour of the light is ‘faded’ depending on the pixel’s distance from the light source – attenuated colour = original colour/distance.

As the models in the scene are relatively close to the light source, it is not apparent that they are affected, however when the troll is moved around the scene, the places where the light hits get dimmer.

Ambient Light

Ambient light is a background level light that makes models visible in the scene without being in direct view of any other light, as shown on the crate. Ambient light is a separate variable that is added to every lighting equation in the scene.

Wiggle Effect

The wiggle effect is the only effect in the scene that is altered through vertex shader manipulation. There are two examples of the wiggle effect.

In the case of the parallax cube, the following code is applied:

float w = sin(modelVertex.position.x \* radians(360.0f) + wiggle);

modelVertex.position.x += 2.0f \* w;

modelVertex.position.y += 2.0f \* w;

This changes the position of the cube in the scene; it moves back and forth in a diagonal. Regarding the sphere, the following code causes the distortion, which moves at the same frequency as the cube but only along one axis:

float w = sin(modelVertex.position.x \* radians(360.0f) + wiggle);

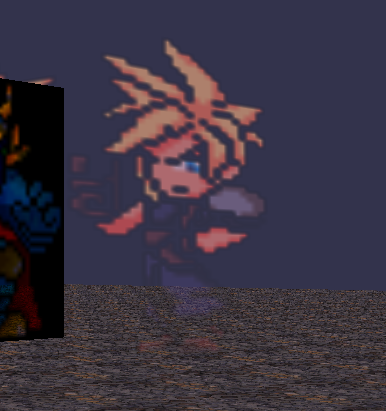
modelVertex.position.x += 2.0f \* w;

Multiplicative Blending

Multiplicative blending is achieved by multiplying the source colour with the destination colour to produce the final colour.

The clearest example of this is when the sphere is viewed through the Golbez image.

Alpha Blending

The alpha blending is the more complicated equation in the blending methods used. The equation is as following:

Final colour = source x source.alpha + destination x (1-source.alpha)

Because the image has a consistent alpha, it is consistently transparent, and it does not affect the scene as clearly as the other methods as a result.

Additive Blending

Additive blending is very similar to multiplicative blending, however instead of multiplying the source colour and destination colour, they are added together to produce the final colour.

This is apparent in the scene via all the lighting effects used.

References

Kennedy, J. and Satran, M. (2018). *Graphics Pipeline - Windows applications*. [online] Docs.microsoft.com. Available at: https://docs.microsoft.com/en-us/windows/desktop/direct3d11/overviews-direct3d-11-graphics-pipeline [Accessed 30 Apr. 2019].

3D Game Engine Programming. (2014). *Introduction to DirectX 11*. [online] Available at: https://www.3dgep.com/introduction-to-directx-11/ [Accessed 30 Apr. 2019].