RTCG Miniproject Pictorial

Felix Bengtvig Krog
Aalborg University,
Copenhagen, Denmark
fkrog22@student.aau.dk
Student No: 20222072

Abstract

This pictorial presents nine exercises done as part of the Real-Time Computer Graphics course of the Medialogy Masters at AAU Cph. Using different real-time graphical engines and 3D software, the following products were created. **1)** A procedurally moving grid of cubes creating visually interesting imagery. **2)** Photogrammetry and Gaussian Splatting of a water bottle. **3)** Photos of said water bottle with different lighting systems used. **4)** A halftone custom lighting model shader. **5)** Post processing effects applied to a virtual camera. **6)** A bezier-curve deformation shader written in HLSL with Lambert lighting. **7)** Real-Time animations. **8)** A mini-virtual production of a Piranha Plant in a Super Mario level. **9)** Point cloud and voxel showcase of the aforementioned water bottle.

Keywords:

Real-Time Computer Graphics; Unity; Photogrammetry; Gaussian Splatting; 3D model; Lighting; Shaders; HLSL; Animation; Virtual Production

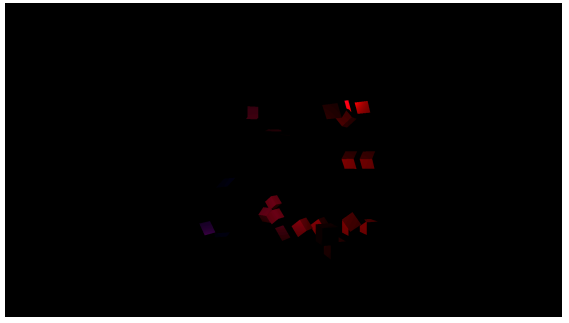
Associated links:

Video, GitHub.

Exercise 1: Grid of Cubes

A grid of cubes is generated and moved around and rotated to create visually interesting imagery. The cubes rotate in one of three directions based on the id in the grid, they use cos for scaling, the light moves in a figure 8, and the raycast gets all objects in the direction of its cast.

Despite being suggested, I did not add the colour changing every frame as it seemed quite epilepsy inducing.

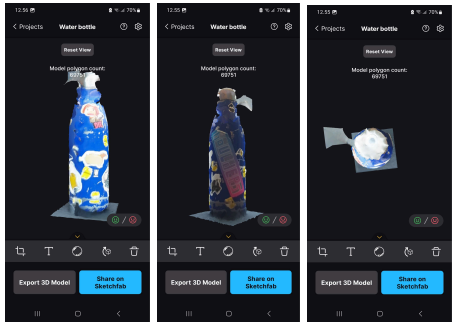


Render of the cube exercise

Exercise 2: Photogrammetry and Gaussian Splatting

The selected object was a water bottle with stickers on it, as it had a simple shape and the stickers would give many areas of

it different patterns, making it generally well-suited for photoscanning. The biggest issues with the photoscans were in the non-sticker covered areas of the bottle. RealityScan was used for photogrammetry, and poly.cam was used for gaussian splatting, utilising the same set of images for both.
After exporting the model, the surrounding environment was removed in blender, and the bottom vertices were connected. The mesh was also decimated to reduce the poly count.



RealityScan Photogrammetry

Gaussian Splatting

Exercise 3: Computer Graphics Lighting

After the water bottle was imported into Blender, a simple showcase of it was created, using two light sources, two matte walls, and a metallic ground surface. The same scene was imported into Unity, with the exact same positionings. In Unity, the reflectiveness of the ground surface is not visible, as the project utilises the URP without any raytracing. The scene could have been made to more closely resemble the Blender counterpart if a reflection probe had been baked, which would have made it a cheaper version of the same scene without much loss.

Photoshoot of Water Bottle in Blender

Same photoshoot imported into Unity

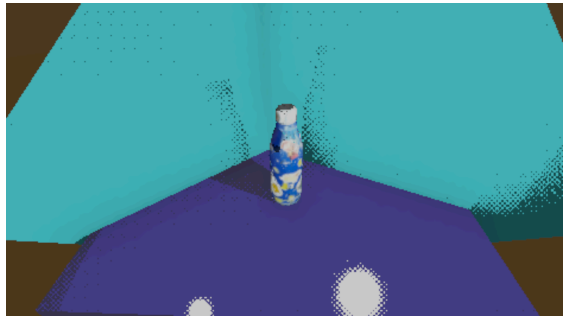
Exercise 4: Shader Graph

Inspired by halftone shading found in comics and some games, I chose to create a halftone lighting shader.

Halftone shadows in Kirby

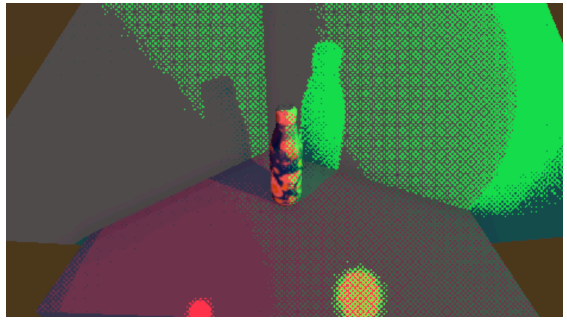
This custom lighting model (CLM), being inspired by Ned Makes Games CLM [1], uses the Forward rendering path of URP, allowing for up to 8 additional light sources affecting a game object at a time. Cast shadows from additional lights are also

supported, and will have the growing circle pattern of the halftone shading style.



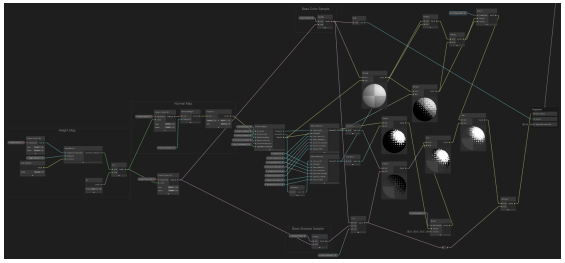
The water bottle photoshoot using the Halftone CLM

The CLM also allows for multiple differently coloured lights affecting the same game object, and will blend them using the halftone style as well.

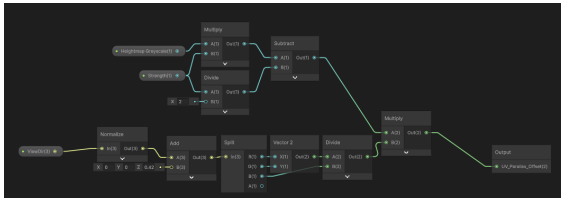


The water bottle photoshoot with green and red lights and the Halftone CLM

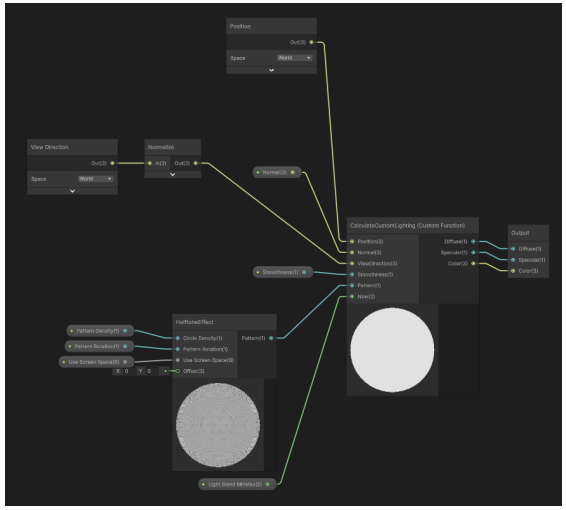
The shader behind this is quite long, and as such it will only be described in broad terms in this pictorial. Textures are given to the shader, which then have tiling, offset, height map, and normal map calculations done. A HLSL script gathers information about the lights affecting the game object and outputs the diffuse and specular intensity of the light, as well as the colour of it, following the Phong reflection model. The diffuse and specular separately have their halftone patterns calculated for them, as the specular is toggleable depending on whether the material should act reflective or not. After the patterns have been applied, the colour of the light for the pixel given by the diffuse is multiplied by the colour given by the texture. If specular light is enabled, it is added after this as the final step.



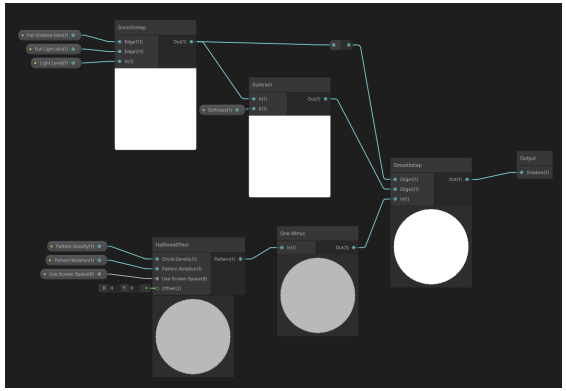
Main Halftone shadergraph



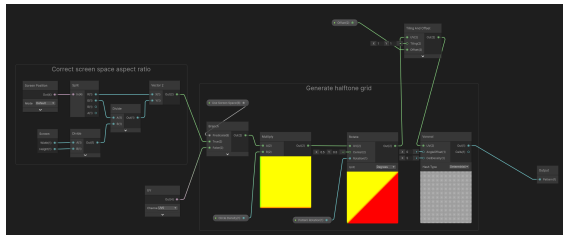
Heightmap subshader



Custom Lighting subshader



Halftone Shadow subshader. Applies the halftone pattern based on intensity of light



Halftone Effect subshader. Calculates the halftone pattern in screen space using voronoi.

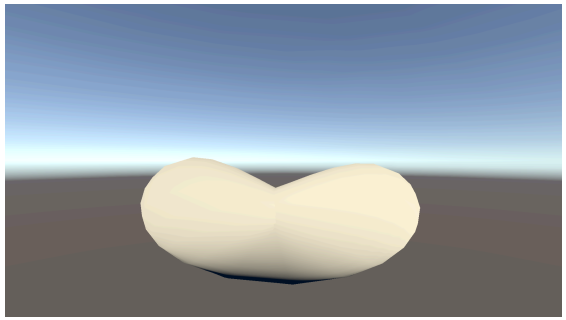
Exercise 5: Computer Graphics Camera

As my only available computers are a laptop with an integrated graphics card and another computer with a graphics card from 2012, I cannot run HDRP without experiencing a plethora of issues. As such, this was done in the URP, which does mean that the camera settings themselves were not effective, instead only the post-processing effects changing the rendered image. The water bottle photoshoot was once again utilised.
For this exercise, the camera was zoomed further out, but used a lens with a high focal length to still keep the water bottle large in the final image. Depth of field was added (also used in exercise 8) to slightly blur behind the water bottle, and colour curves and bloom were utilised to make the specular reflection on the bottle more distinct, and the environment more moody.

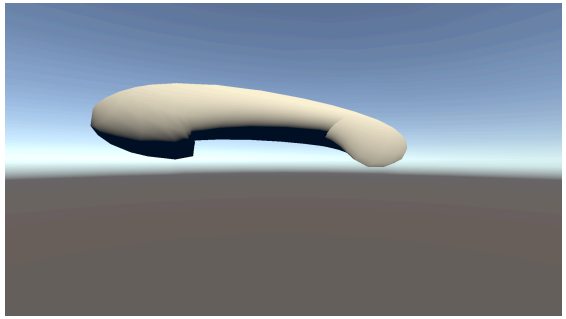
The water bottle photoshoot with post processing effects.

Exercise 6: Shader Code

For this exercise, I took the Bezier Curve shader from shaderslab.com [2], and found that it was very lacking and not something I wanted. So I rewrote it entirely to not make assumptions about the geometry of the model, use cubic bezier curves instead of quadratic, and also alter the normals of the affected game object. On top of this vertex shader, I applied a simple Lambert lighting shader from the Unity Shaders github [3], as I found it to be a great tool to visualize the normals, and I didn't just want to colour the normals based on the direction.



A sphere having its vertices transformed by the shader.



Another sphere transformed along a different bezier curve by the shader.

The shader itself calculates a bezier-curve in local space, and any point on the model that can be projected onto a linear line between the start and end points of that bezier curve have their positions and normals transformed.

```
Properties
{
    _MainTex ("Texture", 2D) = "white" {}
    _AmbientLight("Ambient Light", Color) = (0.0,0.075,0.15, 1)
    _StartPoint ("Start point", vector) = (-0.5, 0, 0, 1)
    _EndPoint ("End point", vector) = (0.5, 0, 0, 1)
    _DirectionOne ("Direction 1", vector) = (0, 0, 0, 1)
    _DirectionTwo ("Direction 2", vector) = (0, 0, 0, 1)
}
```

```
// Lerps for a float3
float3 Lerp3D(float input)
{
    return float3(lerp(_StartPoint.x, _EndPoint.x, input),
                  lerp(_StartPoint.y, _EndPoint.y, input),
                  lerp(_StartPoint.z, _EndPoint.z, input));
}

// Finds out how far along the model's bezier-curve this is using a normalised dot product
float InverseLerp3D(float3 inputVertex)
{
    return dot(inputVertex - _StartPoint.xyz, _EndPoint.xyz - _StartPoint.xyz) / length(_EndPoint.xyz - _StartPoint.xyz);
}

// Gets a point on a cubic bezier curve based on T
float3 BezierPoint(float input)
{
    float mt = 1 - input;
    return pow(mt, 3) * _StartPoint.xyz + 3 * pow(mt, 2) * input * _DirectionOne + 3 * mt * pow(input, 2) * _DirectionTwo + pow(input, 3) * _EndPoint;
}

// Gets the tangent of the bezier curve based on T
float3 BezierTangent(float input)
{
    float mt = 1 - input;
    return 3 * pow(mt, 2) * (_DirectionOne - _StartPoint) + 6 * (mt) * input * (_DirectionTwo - _DirectionOne) + 3 * pow(input, 2) * (_EndPoint - _DirectionTwo);
}

// Gets the acceleration of the bezier curve based on T
float3 BezierAcceleration(float input)
{
    float mt = 1 - input;
    return 6 * (mt) * (_DirectionTwo - 2 * _DirectionOne + _StartPoint) + 6 * input * (_EndPoint - 2 * _DirectionTwo + _DirectionOne);
}
```

```
v2f vert (appdata_base v)
{
    v2f o;

    // Returns a 0-1 number describing how far along the line the projected vertex is, clamping above and below values
    float distanceOnLine = saturate(InverseLerp3D(v.vertex.xyz));

    // Figure out where in the world this 0-1 would be
    float3 bezierPoint = BezierPoint(distanceOnLine);

    float3 regularLinePoint = Lerp3D(distanceOnLine);


    // Figure out the new normal based on the bezier curve
    float3 bezierTangent = BezierTangent(distanceOnLine);
    float3 binormal = normalize(cross(bezierTangent, BezierAcceleration(distanceOnLine)));

    bezierTangent = normalize(bezierTangent);

    float3 pointNormal = normalize(cross(binormal, bezierTangent));

    // Make transformation matrix from the point normal, the bezier tangent, and their cross product
    float3 bezierLastBasisVector = normalize(cross(bezierTangent, pointNormal));

    float3x3 normalTransformer = { bezierTangent.x, pointNormal.x, bezierLastBasisVector.x,
                                   bezierTangent.y, pointNormal.y, bezierLastBasisVector.y,
                                   bezierTangent.z, pointNormal.z, bezierLastBasisVector.z};

    // Offset vertex from the bezierPoint with the normal taken into account

    v.vertex.xyz = bezierPoint + mul(normalTransformer, (v.vertex.xyz - regularLinePoint.xyz));

    o.pos = UnityObjectToClipPos(v.vertex);
    o.uv = TRANSFORM_TEX(v.texcoord, _MainTex);
    o.normal = UnityObjectToWorldNormal(mul(normalTransformer, v.normal));
    return o;
}
```

```
fixed4 frag (v2f i) : SV_Target
{
    // Direct light
    float3 lightSource = _WorldSpaceLightPos0.xyz;
    float lightFalloff = max(0, dot(lightSource, i.normal)); // 0f to 1f
    float3 directDiffuseLight = _LightColor0 * lightFalloff;

    // Composite
    float3 diffuseLight = directDiffuseLight + _AmbientLight;
    float3 result = diffuseLight * tex2D(_MainTex, i.uv);

    return float4(result, 1);
}
```

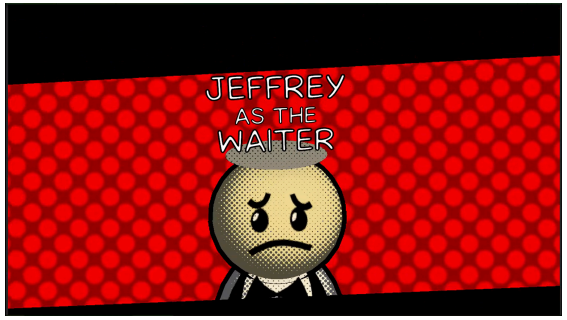
The HLSL shader combining bezier curve deformation and Lambert lighting.

Exercise 7: Real-Time Animation

I combined this Real-Time animation with my semester project, doing a cinematic intro cutscene as I have done multiple times in the past. The video can be seen here.

A shadowed figure throwing five pictures onto a blueprint map.

This cutscene uses mixamo animations, and self animated inverse- and forwards kinematics, both of them in conjunction with the mixamo animations. The cutscene also ends with a seamless transition into gameplay, an affordance given thanks to the cutscene being real-time and not prerendered. The cutscene being real-time also means less storage requirements, as no mp4 or other video format needs to be saved, but also comes at the cost of requiring more power from the computer, which could affect the quality of it. Another strength of run-time animations and cutscenes is their ability to procedurally adapt to content, such as in a previous semester project of mine where I made a cutscene that introduced all the procedurally generated characters of the game, putting each of them in situations relevant to their randomly chosen roles. One render of that cutscene can be seen here.



The character Jeffrey, who got the Waiter role, being introduced in the opening cinematic of Killer on Board.

Exercise 8: Virtual Production

I set up a Mario level using assets from The Models Resource [4,5], with natural sunlight as one controllable light source, and a lamp box as another. Being limited by the physical light sources I had available, I made the directional light in my Unity scene reflect my room's light in temperature and direction, and made the lamp box shine a light in the same temperature as the screen light of my 3Ds.

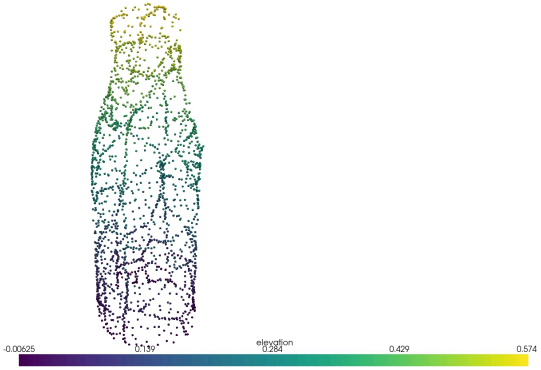
Virtual production of a Piranha Plant figurine in a Super Mario level.

The lighting setup, utilising a 3DS as a light source.

I also built a primitive version of the Piranha Plant figurine in Unity that I made only cast shadows, so that it would look more believably present in the scene.

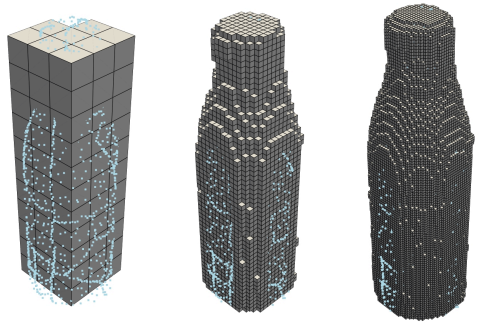
Exercise 9: Analysis and Visualisation of 3D Objects

Using PyVista, I made a point cloud visualisation, colouring the points based on their elevation. This is done on the model with the reduced poly count. The point cloud highlights how certain lines are heavily populated, while other large areas of the flask are quite empty, suggesting room for improvement in the topology of the bottle.



Point cloud of water bottle model.

I also did voxelisations of the bottle to three different precision levels. This gives general information about a simplified form of the bottle, showing that at a very primitive level it can basically be represented by a rectangle.



Voxel representations of my water bottle.

References

[1] NedMakesGames. (2021, August 18). *Creating Custom Lighting in Unity's Shader Graph with Universal Render Pipeline*. Medium. https://nedmakesgames.medium.com/creating-custom-lighting-in-unitys-shader-graph-with-universal-render-pipeline-5ad442c27276

[2] ShadersLab (n.d.). *Deform a model with a Bezier curve*. Shaders Laboratory. http://shaderslab.com/demo-35---deform-with-bezier-curve.html

[3] Adrian Miasik (2022, April 16). *Unity Shaders: Lambert Shader*. GitHub. https://github.com/adrian-miasik/unity-shaders/blob/develop/UnityShaders/Assets/Shaders/HLSL%20-%20Unlit/Lambert.shader

[4] djfox11 (2023, September 7). *1-1: Super Bell Hill - Super Mario 3D World*. The Models Resource. https://models.spriters-resource.com/wii_u/supermario3dworld/asset/345674/

[5] DogToon64 (2024, September 29). *Light Box - Super Mario 3D World*. The Models Resource. https://models.spriters-resource.com/wii_u/supermario3dworld/asset/356878/