GDW Aligned Components Report

By: JayPEG

Members:

Léo Buono (100748457)
Ryan Chang (100745877)
Jonathan Jay (100743575)
Matthew North (100754902)
Matthew McPherson (100757379)
Log-Hei (Jonathan) Leung (100746578)

Part 1: Lighting

Our game had all the basic terms: ambient, emissive, diffuse, specular and rim. It also used deferred rendering.

-Our ambient strength was always high (1 unit), because we didn't want any really dark areas in our game.

-We used emissive materials in a few specific spots, such as the TVs in both the demo map and tutorial map, as well as the glow on the logo. Data stored in specular buffer and emissive buffer, doesn't render to the albedo buffer:

```
//other classes store in material
outSpecs.x = float(!bool(inSpec.x));
outSpecs.y = inSpec.y;
outSpecs.z = receiveShadows;
outSpecs.w = rimLighting;

//colours
outColours.rgb = vec3(inColour + addColour) * outSpecs.x;
outColours.a = inSpec.z;
outEmissive = vec4(inColour + addColour, inSpec.z) * inSpec.x;
```

outSpecs.x stored a emissive flag (0 or 1) for lighting calculations
You can see we use that flag to output to outColour and outEmissive

-Diffuse and specular was based on the type of light (the sun being a global directional light, and smaller lights being points) and used blinn-phong

-Rim lighting was object based, and players were rim lit. stored in specular buffer and done in the ambient phase:

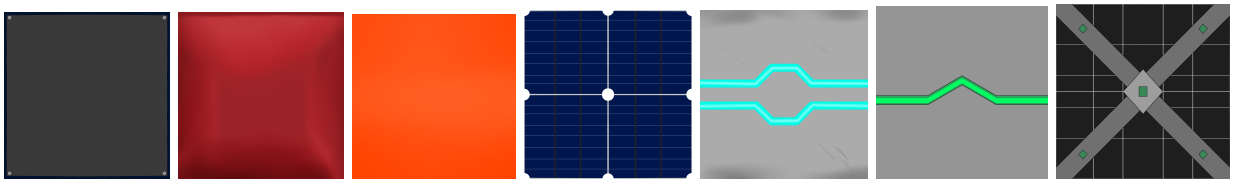
```
lightAccum += (clamp(
    0.5 - dot(
        normalize(u_camPos[camNum] - texture(s_positionTex, inUV).xyz),
        (normalize(texture(s_normalsTex, inUV).xyz) * 2.0) - 1.0
    ), 0.0, 1.0) * float(texture(s_specTex, inUV).w > 0.5)) * rimColour;
```

We had multi-lights done in the directional frag gBuffer shader using blinn-phong

Part 2: Texturing

Our map was made in Hammer, which made UVing it very easy. The map was intentionally divided into colour coordinated sections. The left section was green, the right section was blue and the middle was orange. The colour pallet used was a mix of orange, blue, green and grey. The reason we chose these colours was because we wanted bright contrasting colours.

Texture Samples:



Floor Middle Blue Side Green Side

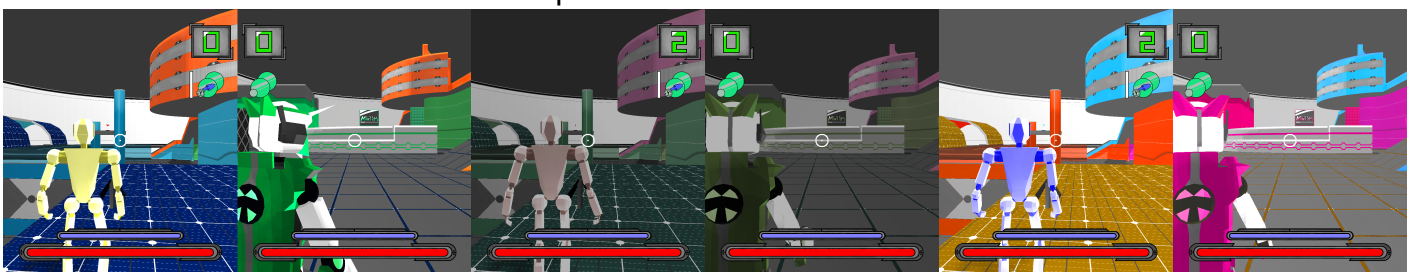
Reference images were taken from solar panels, smooth metal sheets, crates, and padded walls.
Reference image samples:

Note for the rest of the document: most of our rendering was dealt in a class called FrameEffects in Effects/FrameEffects.h, which dealt with transparency, shadows, Gbuffers, etc.

Part 3: Colour Correction

We have a colour correction post effect, which uses LUT cube lookup math to correct the colours. We made 2 cubes (1 used in game, 1 made for this assignment)

Default: “power down” “this”



Default Lower saturation bright

-The “power down” cube was created to be reminiscent of an electrical device shutting down. The colours are intentionally more dull and the brightness is lower.
-In the “this” cube, the colours were intentionally made brighter to liven up the visuals. It was also an experiment to see what other colours could work for the sections of the map instead of green, blue and orange.

Most of this code is the same to the tutorial, other than doing mix(source, cubed, intensity) at the end.

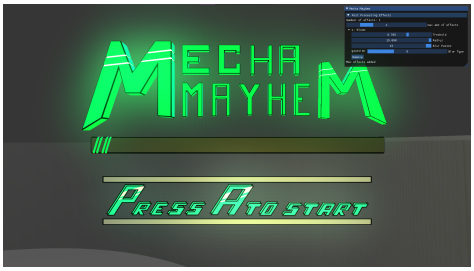
Part 4: Bloom

We have the 3 blur types and bloom in game. We took one of our member’s bloom implementations from the midterm and added it to our project. That implementation had a radius modifier as well, which affects the spread of each of them.

```
//gaussian
if (_blurType == 0) { ... }
//box
else if (_blurType == 1) { ... }
//radial
else if (_blurType == 2) { ... }
```

We used an int to determine the type of blur, to make it easy to deal with, these screenshots will have exaggerated radius to show the effect (look at the ImGui for numbers), all of these on the same scene:

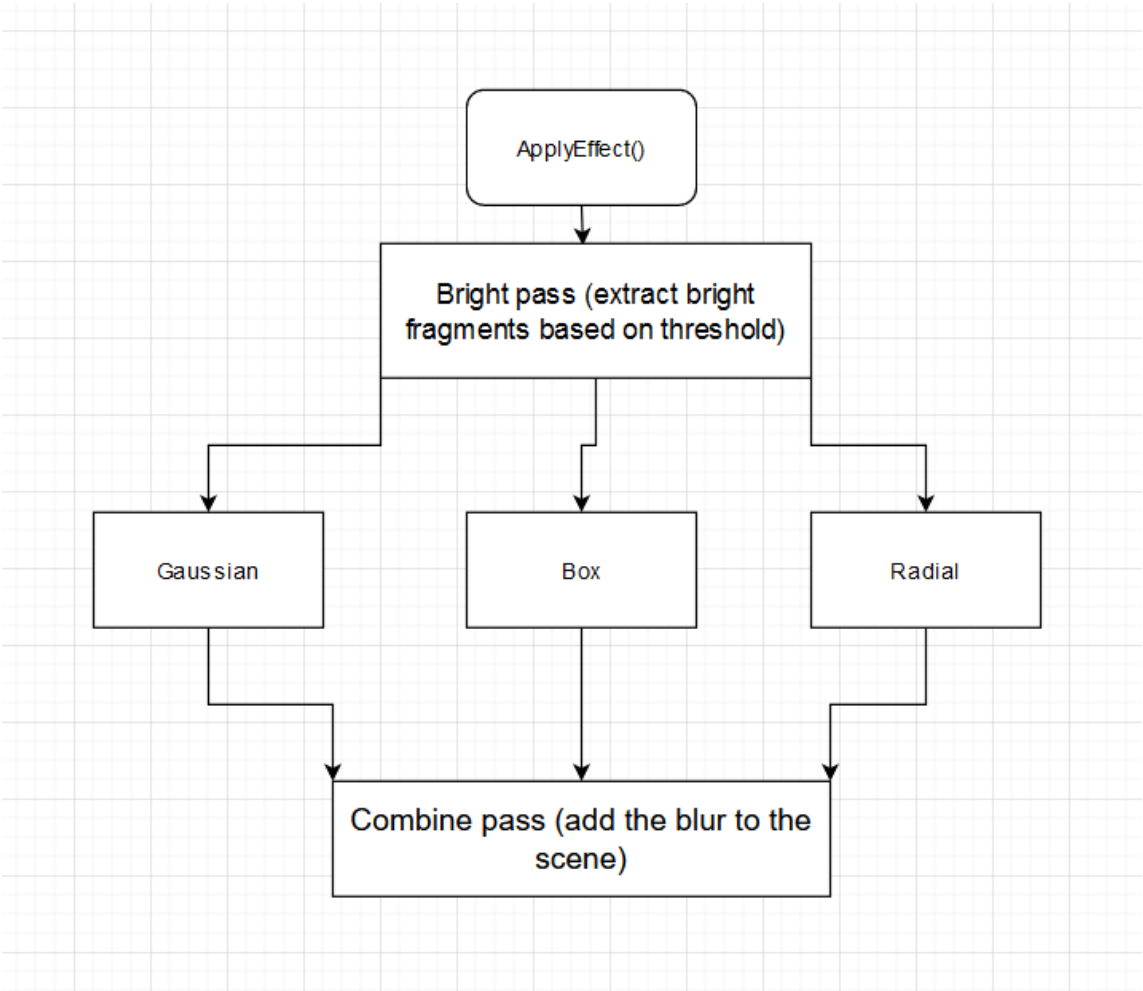
Gaussian



Box

Radial

Flow Diagram:



Part 5: Shadow mapping

We have shadows, the whole map is put under one shadow map, because of the multiplayer aspect and difficulties that come with splitting it up for each. Because of this, there is a lot of peter panning. Our steps were:

-We got shadows in by first writing code for a depth pass (draw all objects, ignoring those that don't cast shadows).

-We wrote a function to generate an ortho shadow VP with offsets.

-We sent all the data to the illumination buffer class.

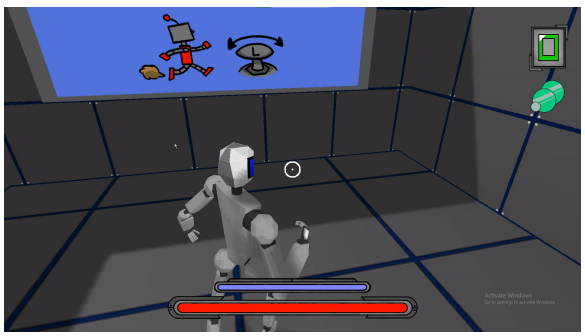
-We used the shadow map implementation discussed in the tutorials, so using the lightspaceviewproj matrix to convert to depth.

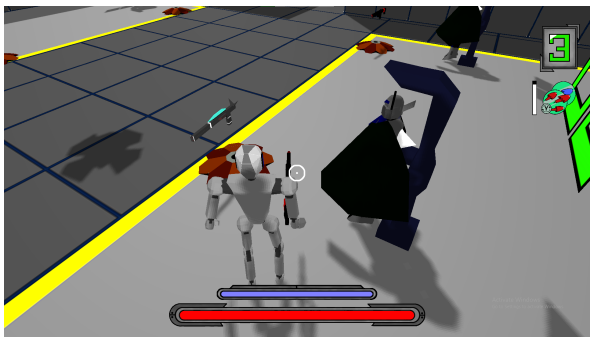
-We used PCF to blur the shadow, which helps mitigate the effect of low texture size

```
float currDepth = projectionCoordinates.z - bias;
float shadow = 0.0;
for (int x = -softRange; x <= softRange; ++x) {
    for (int y = -softRange; y <= softRange; ++y) {
        shadow += float(currDepth >
            texture(s_ShadowMap, projectionCoordinates.xy + vec2(x, y) * texelSize).r
            );
    }
}
return shadow * area;
```

To get shadows, we compare the depth of the fragment with the shadow map (converting position with the shadow VP), and if behind the shadow map, then we shade it (multiply the sun diffuse and ambient by 0). We also added a receive shadow option, which is stored in our specular buffer (either 1 or 0 multiplied with the shadow value, meaning it is either 0 or whatever it stored).

Sample pictures:





Part 6: Pixelation and depth of field

We had both pixelation and depth of field as options. Pixelation we achieved by rendering the previous buffer into a smaller buffer, then back onto a normal sized buffer. This meant we didn't need to write any new shaders (passthrough was used), and only needed to store an extra buffer that was small.

For depth of field, we had 4 steps: extract near, delete near, blur far, and combine, each using their own shader(s)

1. Extract near: using the depth buffer from the gbuffer's buffer, compare to a threshold uniform, then either discard if far, or keep if near.
2. Delete near: we discard if it exists on the extracted near texture. Otherwise keep
3. Blur: we used two shaders, horizontal and vertical. In both, we did 9 passes and used the alpha value to ignore bad texels (multiply the pass by float(alpha > 0.31), which removed bad texels, at the end, do: frag_colour / frag_Colour.a to restore levels). We had a for loop to control how many passes we performed.
4. Combine: paste the near buffer from earlier over the blurred pass.