CS4247 Cheatsheet
by Yiyang, AY22/23

Introduction

OpenGL Shaders

Vertex Shader

Fragment Shader

Texture Mapping

OpenGL Cube Map

Shaders

Cartoon Style Shading

Procedural Bump & Normal Mapping

Procedures for Procedural Bump Mapping:

1. At each vertex, transform normal & tangent to eye space
2. Interpolate eye-space normal & tangent to fragments
3. At each fragment, compute **Binormal vector** & compute tangent-space **Perturbantion Vector** based on texture coordinates
4. Transofmration perturbantion from tangent space to eye space and use it for lighting computation

Binormal Vector - unit vector perpendicular to both normal & tangent vectors.

In **Normal Mapping**, use a normal map where each texel contains 3 perturbation vector components in RGB channels.
Note: Need to encode from $[-1, 1]$ to $[0, 1]$ for each value read.

Refraction & Reflection Mapping

Procedures:

1. Set up an **Environment Map** (e.g. cubemap)
2. Compute the refracted ray of the view ray
3. Use (interpolated) refraction ray direction to access env.map at fragment shader

Note: [1] We only compute refraction at the first surface (entering the object) and ignore when the ray exits the object since it is too complicated. [2] Refraction mapping is similar to reflection mapping, but need to account for Fresnel Effects and Chromatic Aberration.

Fresnel Effects

Fresnel Effects - Refraction depends on viewing angles. Look at an object from top (side), the surface looks shinnier with more refraction (less shiny and more reflection).
Fresnel's Equation determines the **ratio** of reflected and transmitted light energy from a perfect surface:

$$F = \frac{1}{2}\left(\frac{\sin^2(\phi - \theta)}{\sin^2(\phi + \theta)} + \frac{\tan^2(\phi - \theta)}{\tan^2(\phi + \theta)}\right)$$

, where ϕ and θ are angles of incidence and refraction, and $\mu = \sin\phi / \sin\theta$ is the refractive index of the material.
Schlick's Approximation approximates Fresnel's Equation:

$$F = f + (1-f)(1 - V \cdot N)^5, \text{ for } f = \frac{(1 - n_1/n_2)^2}{1 + n_1/n_2)^2}$$

Chromatic Aberration

Chromatic Aberration - Realistic (flawed) cameras produce images where different colour rays do not converge on the same point. Instead, there are slightly different extents of refraction for different colour components.
Solution: Use different refractive indices for RGB channels.

FBO & Real-Time Shadow

Multi-Pass Rendering - render 3D scenes for multiple times (passes), and combine the multiple rendered images to synthesize the final frame.

Framebuffer Object

Framebuffer Object (FBO) - Framebuffers created in OpenGL for storing rendering outputs, that can be non-displayable.
Each FBO contains many **Attachment Points**, i.e. rendering destinations, with two types of FBO attachable images:

1. Texture images, render to texture
2. Render buffer images, offsceen rendering

Note: **Mutliple Render Targets** (MRT) enables a fragment shader writes to (the same position of) multiple FBOs in one pass at the same time.

Real-Time Shadow

Point light sources produce shadows with **hard boundaries**, while extended light sources give **soft bounaries**, with **Umbra** and **Penumbra**.
Note: Real-time shadow rendering techniques assume **point light sources** and generate **Geometric Shadows** (shadows with correct shapes and arbitrary intensities).

Shadow Volume Method

Idea: An occluder casts a shadow volume. After determining the occlusion boundaries for each occluder, use stencil buffer to keep trace of how many are occluding the current pixel.
Stencil Buffer has the same resolution as colour buffer, and each stencil "pixel" is an integer counter.

Procedure for **Depth Pass Method**:

1. Find silhouette edges
2. Extend silhouette edges from light source to form shadow volume polygons
3. Render the scene as if it were completly in shadow
4. Disable writes to the depth and color buffers
5. Draw all **front faces** of shadow volume, and increment stencil buffer by 1 on **depth pass**
6. Draw all **back faces** of shadow volume, and decrement stencil buffer by 1 on **depth pass**
7. Render the scene again as if it were completely lit, using the stencil buffer to mask the shadowed areas

Analysis: [1] It draws only on pixels whose stencil value is 0. [2] **Fail** if the viewpoint is in a shadow volume or some shadow volume polygons are clipped by the near plane.

Procedure for **Depth Fail / Carmack's Reverse Method**:
Same steps as Depth Pass Method except for how it draws (i.e. Step 5 and 6)

- Draw all **back faces** of shadow volume, and increment stencil buffer by 1 on **depth fail**
- Draw all **front faces** of shadow volume, and decrement stencil buffer by 1 on **depth fail**

Shadow Mapping Method

Idea: Objects are in shadow when their depth map values when viewing from the light source are smaller.
Procedures:

1. Render the scene using the light source as viewpoint
2. Save the depth buffer known as **Shadow Map**
3. Clear the framebuffer
4. Render the scene from camera's viewpoint: For each fragment, transform it to the "light space" and compare its "light space" z-value with shadow map z-value. If "light space" z-value is larger, the fragment is in shadow & lit with only ambient light. Otherwise, the fragment is not in shadow & is fully lit.

For each point $\vec{p}_M$ in modelling coordinates, its shadow mapp coordinates $\vec{p}_L$ is

$$\vec{p}_L = B \cdot P_L \cdot V_L \cdot M \cdot \vec{p}_M$$

, where M, V_L, P_L are the **Modelling Matrix**, **Light's View Transformation Matrix**, **Light's Projection Matrix**, and B is defined as

$$B = S(\frac{1}{2}, \frac{1}{2}, \frac{1}{2})\ T(1,1,1) = \begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Issue 1: Shadow Acnes & Self-Shadowing
Solution: [1] Subtract a tolerance value from `ShadowCoord.z`. [2] Offset the scene backwards when generating the shadow map using `glPolygonOffset()`.

Issue 2: Jaggies at shadow boundaries
Solution: Use **Percentage Closer Filtering** to smoothen, which averages depth comparison results in a small neighbourhood in the shadow map.

Post-Rendering Processing

Post Rendering Image Processing ~Further processing of images produced by 3D rendering, in order to produce final frame.

Edge Detection

It uses 2D digital convolution with **Sobel Operator**, which is defined as:

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \; S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Procedures for edge detection:

1. Apply convolution to the image with Sober Operator kernels. Let results be s_x and s_y.
2. Compute for each pixel $g = \sqrt{s_x^2 + s_y^2}$.
3. If a pixel has g above a threshold, consider it an edge pixel.

Window Coordinates & Texel Coordinates

Window Coordinates: A fragment's 2D position relative to the window. Consider **the center of** the fragment. For bottom-leftmost pixel, it is `gl_FragCoord.x == gl_FragCoord.y == 0.5`.
Texel Coordinates: Indices of the texture map (as a 2D array). For bottom-leftmost toxel, it is $(0, 0)$. Used in `texelFetch()`.
Texture Coordinates: Normalised coordinate values of the texture map $\in [0, 1]^2$. For the bottom-leftmost corner, it is $(0.0, 0.0)$. To calculate the coordinates of the center of a texel, average the coordinates of the corners. Used in `texture()`.
Note: For all coordinate systems above, x & y-coordinate correspond to width from left to right, & height from bottom to top.

Gaussian Blurring

2D **Gaussian Kernel / Filter** is defined as:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$
$$= G(x)G(y) \; , \; G(x) = \frac{1}{2\pi\sigma^2} e^{-x^2/2\sigma^2}$$

It is **separable** as we can express the operation as two 1D Gaussian operations.
2D Digital Convolution for Gaussian Blurring is:

$$C_{lm} \leftarrow \sum_{i=-4}^{4} \sum_{i=-4}^{4} \frac{G(i,j)}{k} C_{l+i,m+j}$$
$$\leftarrow \sum_{i=-4}^{4} \frac{G(i)}{k} \sum_{j=-4}^{4} \frac{G(i,j)}{k} C_{l+i,m+j}$$
$$\text{for } k = \sum_{i=-4}^{4} G(i)$$

Note: All Gaussian weights must sum to 1.
Since it can be done using two 1D convolution, we can perform it in 3 passes efficiently:

1. Bind to 1st FBO & render 3D scene normally to a texture.
2. Bind to 2nd FBO & apply **vertical** Gaussian blur to Pass 1 texture & write to another texture.
3. Bind to 3rd FBO & apply **horizontal** Gaussian blur to Pass 2 texture & write to output colour buffer.

GLSL Coding

Math

`genType sqrt(genType x)` - To compute the square root of input.
`genType min(genType x, genType y)` - To compute the min of two inputs. Similarly for `min(x,y)`.
`genType pow(genType x, genType y)` - To compute the power x^y.
`genType dFdx(genType p)` - To compute the partial derivative of input wrt. x. Similarly for `dFdy()`. `float dot(genType x, genType y)` - To compute the dot produce. Similarly for `cross()`.
`float length(genType x)` - To compute the L2 length of the vector.
`genType normalize(genType v)` - To normalize the input vector.
`genType reflect(genType I, genType N)` - To calculate reflected direction vector.

Fragment Shader

`gvec4 texture(sampler, P)` - To sample texture from the sampler using texture coordinates.
`gvec4 textureProj(sampler, P)` - To first compute the query vector of P using perspective division, then sample texture.
`gvec4 textureProjOffset(sample, P, offset)` - To compute a texture lookup with projection and offset.
`gvec4 texelFetch(sampler, P, lod)` - To sample texture from sampler using texel coordinates. Usually `lod = 0` for Level of Details.
Note: Return types, `P` & `offset` can all be vectors of size 1 4 depending on params.