CS4247 Cheatsheet
by Yiyang, AY22/23

Introduction

OpenGL Shaders

Vertex Shader

Fragment Shader

4. Texture Mapping

OpenGL Cube Map

5. 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.

6. 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 completely 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

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

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.

7. 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 texel, 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:

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

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

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

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.

8. Deferred Shading

Deferred Shading - Techniques to avoid applying complex shaders on fragments not appeared in the final image, based on multi-pass rendering.

1. 1st Pass uses a simple shader to render scene into off-screen buffers with depth info, stored in **Geometry Buffer / G-Buffer** for shading computations.
2. 2nd pass applies frag.shader on G-buffer data to compute for each output pixel.

Note: Writing to G-buffer can be (time & memory) expensive, so deferred shading is used only when direct computation very expensive. Usually used when: 1) High depth complexity / many overdraw, 2) Almost fill entire viewport, &/or 3) Expensive shading.

Screen Space Techniques

Ambient Occlusion - Technique to simulate illumination on **diffuse** surfaces by a **uniform hemispherical area light source**.
Procedures: [1] For each surface point, pre-computing **Visibility / Accessibility** factor, proportion of hemisphere centered about the surface normal visible from the point. [2] Use the factor to modulate direct diffuse illumination during rendering.
Issues: [1] Accessibility factor at each vertex can be computed by shooting rays around, expensive! [2] Pre-computed factors cannot be used for dynamic scenes.

Solution for expensive AO: **Screen-Space Ambient Occlusion** (SSAO) ~Apply deferred shading and compute accessibility only for output fragments based on depth map info.

9. Ray Tracing

In modern OpenGL, ray tracing is performed in **fragment shader**.

Ray Tracing Acceleration

Bounding Volume Methods

(Naive) Bounding Volume ~Use a simple shape to enclose each complex object. Quick reject if ray does not intersect the volume.
Note: Trade-off between intersection efficiency & tightness for rejection rate.

Bounding Volume Hierarchy ~Improvement to Bounding Volume method by organising volumes into hierarchies.
Note: Good hierarchies usually constructed manually.

Spatial Subdivision Methods

Spatial Subdivision Method - Subdivide 3D space into regions & associate each region with a list of objects that occupy (fully or partially) the region. When a ray is traced into a region, query the object list and perform intersection tests with objects.
Choices for subdivision shape: 1) Uniform Grid, 2) Octree, 3) Binary Space Partitioning (BSP).

Octree ~Each cube region is conditionally & recursively subdivided into 8 equal subregions.
Two subdivision schemes

1. Subdivide a cell if occupied by more than 1 object
2. Subdivide a cell if occupied, until maximum depth

Distribution Ray Tracing

Distribution Ray Tracing - For each pixel, shoot multiple random rays to perform ray tracing. At each intersection, **randomly perturb** reflection, refraction & shadow rays according to some distribution.
Analysis: Two sources of randomness: 1) shoot random rays, 2) add random perturbations during interaction.

10. Local Reflection

Local Reflection ~Consider relationship between a light source, a surface point & a view point.
Global Illumination ~Consider all light sources & surfaces.
Note: [1] Local reflection involves no interaction with other objects. [2] Shadow is global illumination.

Local reflection improves material appearances by improving reflection, **Bi-directional Reflection Distribution Function** (BRDF), & transmission, **Bi-directional Transmission Distribution Function** (BTDF).

Radiometry Basics

Radiometry - The study of physical measurement of light / radiation energy.

- **Radiant Power / Radiant Flux**, Φ - Total energy flows from/to/through a surface per unit time. (Unit Watts, W)
- **Irradiance**, E - Incident (incoming) radiant power on a surface per unit surface area, $E = \frac{d\Phi}{dA}$. (Unit W/m^2)
- **Radiant Exitance**, M / **Radiosity**, B - Exitant (outgoing) radiant power from a surface per unit surface area, $M = B = \frac{d\Phi}{dA}$. (Unit W/m^2)
- **Radiance**, L - Radiant power per unit projected surface area per unit solid angle. (Unit $W/(sr\ m^2)$)

$$L = \frac{d^2\Phi}{d\omega dA \cos\theta}$$

Notes: [1] L invariant along straight paths. [2] Sensors' response proportionate to L incident upon them.

Bi-directional Reflection Distribution Function

Bi-directional Reflection Distribution Function - Specify the amount of incident light reflected in an exitant direction.

$$\begin{aligned} \text{BRDF, } f_r(x, \Psi \to \Theta) &= \frac{dL(x \to \Theta)}{dE(x \leftarrow \Psi)} \\ &= \frac{dL(x \to \Theta)}{L(x \leftarrow \Psi)(N_x \cdot \Psi) d\omega_\Psi} \end{aligned}$$

, where x is surface point, $\Psi = L, \Phi = V$ light source & sensor.
Note: Radiosity quantities vary with wavelength. **Therefore we need a separate BRDF for each of RGB components.**

BRDF Properties

- **Helmholtz Reciprocity** - BRDF is symmetric wrt. the incident and reflected directions.

$$f_r(x, \Psi \to \Theta) = f_r(x, \Theta \to \Psi)$$

- Energy Conservation - Total power reflected in all directions must be equal or less than incident power.

$$\int_{\Omega_x} f_r(x, \Psi \to \Omega)(N_x \cdot \Theta) d\omega_\Theta \leq 1, \ \forall \Psi$$

Cook-Torrance Reflection Model

Cook-Torrance / Torrance-Sparrow Reflection Model - A physically based **specular reflection** model that models surfaces using **Microfacets** which are simplified as symmetric V-shaped grooves.

$$f_r(x, \Psi \leftrightarrow \Theta) = \frac{D(\alpha)GF}{(N \cdot \Psi)(N \cdot \Theta)\pi} + \frac{k_d}{\pi}$$

Note: Diffuse one can be calculated using other methods & overall effect is $sR_s + dR_d$ subject to $s + d = 1$.
Analysis: [1] Assume microfacets are perfectly clearn. [2] CTRM BRDF is isotropic, & cannot model anisotropic behaviours for objects like brushed metals, clothes or fur.
Isotropic - BRDF is independent of the azimuth angle ϕ of incident light (look the same around $360°$ left to right).

$$f_r(x, (\theta_o, \phi_i) \leftrightarrow (\theta_o, \phi_o)) = f_r(x, (\theta_o, 0) \leftrightarrow (\theta_o, \phi_o - \phi_i))$$

A BRDF that is not isotropic is **Anisotropic**.

Four components

1. Statistical distribution of microfacets' orientation $D(\alpha)$
2. Shadowing & Masking effects, G
3. Glare effect, $(N \cdot V)$
4. Fresnel term, F

CTRM - Microfacet Distribution

Model proportion of microfacets facing a given direction, $D(\alpha)$, typically using a simple **Normal Distribution**:

$$D(\alpha) = k \exp\left(-(\alpha/m)^2\right)$$

, where α is angle between N and H, and m controls mean surface roughness.

CTRM - Shadowing & Masking

Some light might be trapped or intercepted, leading to **Masking**, cannot go out along reflected, &/or **Shadowing**, incident light being blocked.

$$G = \min\{1, G_s, G_m\}$$
$$G_m = 2(N \cdot H)(N \cdot V)/V \cdot H$$
$$G_s = 2(N \cdot H)(N \cdot L)/V \cdot H$$

CTRM - Glare Effect

Glare Effect - When angle between the view vector and the mean surface normal increases, an observer sees more microfacets.
Implementation: [1] Modelled with the term $1/N \cdot V$. [2] Countered by masking effect if viewing direction almost parallel to mean surface.

11. Global Illumination

Rendering Equation

Hemispherical Formulation

$$L_{\text{ref}}(\mathbf{x}, \omega_{\text{ref}}) = L_e(\mathbf{x}, \omega_{\text{ref}}) + \int_s \rho(\mathbf{x}, \omega_{\text{in}} \to \omega_{\text{ref}}) L_{\text{in}}(\mathbf{x}', \omega_{\text{in}}) \cos\theta_{\text{in}} d\omega_{\text{in}}$$

Area Formulation:

$$L_{\text{ref}}(\mathbf{x}, \omega_{\text{ref}}) = L_e(\mathbf{x}, \omega_{\text{ref}}) + \int_s \rho(\mathbf{x}, \omega_{\text{in}} \to \omega_{\text{ref}}) L_{\text{in}}(\mathbf{x}', \omega_{\text{in}}) g(\mathbf{x}, \mathbf{x}') \cos\theta_{\text{in}} \frac{\cos\theta_0 dA}{\|\mathbf{x} - \mathbf{x}'\|}^2$$

, where $g(\mathbf{x}, \mathbf{x}')$ is the **Visibility Function**, & $d\omega_{\text{in}} = \frac{\cos\theta_0 dA}{\|\mathbf{x}-\mathbf{x}'\|}^2$ projected area of the differential surface region visible in direction ω_{in}.
Analysis: [1] View-independent. [2] Recursive & complicated, cannot be evaluated analytically, so numeric methods like Monte Carlo are used.

Monte Carlo Integration

Monte Carlo Integration - A method of evaluating an integral by generating random samples of the integrand & taking the average. Sampling techniques determine accuracy / efficiency of Monte Carlo estimates:

- **Uniform Sampling**. Error $\propto 1/\sqrt{N}$.
- **Stratified Sampling** - Subdivide domain into non-overlapping regions, & take one random sample per region. Error $\propto 1/N$.
- **Importance Sampling** - Sample the integrand with probability similar to the value of the integrand. Error depends on choice of probability function chosen.

Ray Tracing Models

Light Transport Model ~Consider reflection as (totally) specular or (totally) diffuse for surface-to-surface interaction. Use regex-liked notations to describe light transportation.

- `L` light source; `E` eye; `S` specular reflection; `D` diffuse.
- For any event `e`: `(e)+` one or more; `(e)*` zero or more; `(e)?` zero or one; `(e|f)` either one.

Model: Whitted Ray Tracing

Analysis: [1] View-independent. [2] Simulate `LD?S*E`. [3] A hybrid method: both global & local model, and global parts only covers **pure specular-specular interaction**.

Model: Distribution Ray Tracing

Analysis: [1] View-dependent. [2] Simulate `LD?S*E`, but all specular paths are calculated, not just in mirror reflection direction.

Model: Path Tracing

Procedures: 1) For each pixel, shoot multiple random primary rays. 2) At each intersection, shoot any one secondary ray. 3) Terminate either when the ray hits nothing or based on **Russian Roulette Technique** for absorption, where randomly at an intersection, absorb / despawn with a probability proportional to material absorption.
Note: Linear complexity, as one secondary ray every time.
Analysis: [1] View-dependent. [2] Simulate `L(D|S)*E`. [3] Complete global illumination. [4] Linear complexity but high computational cost, need many rays for indirect illumination effects like caustics.

Model: Two-Pass Ray Tracing

Procedures

- First Pass, **Light Pass**: Rays are shot from the light sources & followed through transparent & specular objects until they hit a diffuse surface. Cache light energy on diffuse surfaces.
- Second Pass, **Eye Pass**: Use conventional ray tracing. Terminate on the diffuse surface & uses the stored energy in the illumination map.

Analysis: [1] First pass view independent, second view-dependent. [2] Simulate `LS+DS*E`. [3] More efficient than path tracing.

Model: Photon Mapping

Photon Mapping ~Emit photons towards all surfaces, & store when hitting a diffuse surface. Randomly determine to reflect, transmit or absorb. For each photon, store [1] incoming direction, & [2] light power, using **KD-Tree** based on 3D location.
Procedsures based on Two-Pass Rendering

- First pass, **Photon Tracing**: Build photon map structure by tracing photons from lights through the model.
- Second Pass, **Rendering**: Render model from eye using photon map information.

Model: Radiosity + Path Tracing

Radiosity ~Discretise scene into patches, consider interaction between patches. Solution expressed in terms of a constant radiosity for each patch.
Analysis: [1] View-independent. [2] Simulate `LD*E`.

Radiosity with Path Tracing: Trace rays from light sources & cache the irradiance on diffuse surfaces. Radiosity solution uses the cached irradiance as patch emission.
Analysis: Simulate `LS*D*S*E`.

12. Radiosity

Radiosity ~An approximation algorithm for modelling diffuse-to-diffuse interaction in the scene by discretising the scene into patches & calculating diffuse-to-diffuse interaction between patches. Based on Rendering Equation:

$$B_i = E_i + R_i \int_j B_j F_{ij}$$
$$\approx E_i + R_i \sum_{j=1}^{n} B_j F_{ij}$$

Intuitively, it is radiosity × area = emited power + reflected power.
Note: Iterative / numerical methods are used to solve the system of linear systems for B_i. [2] E_i, R_i are **wavelength-dependent**, a separate set of equations for each of RGB.

Solving Linear Equations

Algorithm: Gaussian Elimination

Analysis: [1] Able to get exact solution but unnecessary for radiosity. [2] Time complexity $O(n^3)$.

Algorithm: Jacobi Iteration / Gauss-Seidel Method

Jacobi Iteration - For a system of linear equations $A\mathbf{x} = E$,

1. Initial approximation $x_i^{(0)} = E_i / a_{ii}$
2. Compute $x_i^{(k+1)}$ iteratively using:

$$x_i^{(k+1)} = (E_i - \sum_{j=1}^{n} a_{ij} x_j^{(k)}) / a_{ii},\ \forall i$$

3. Stop iteration if $|x_i^{(k+1)} - x_i^{(k)}| < \epsilon$ for pre-set threshold $\epsilon > 0$.

Gauss-Seidel Method - It improves Jacobi Iteration by using previous unknowns' estimates from the current iteration for all subsequent unknowns:

$$x_i^{(k+1)} = (E_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i}^{n} a_{ij} x_j^{(k)}) / a_{ii},\ \forall i$$

Form Factors

Form Factor, F_{ij} - proportion of energy leaving surface i & striking surface j directly over total energy that leaving surface i.
Form factor for patches can be approximated by sum over A_i.

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi r^2} dA_j dA_i$$

$$F_{dA_i A_j} \approx \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi r^2} dA_j$$

Note: $\int F_{ij} dA_j = 1$.

Nusselt Analogue & Hemicube

Nusselt Analogue - The form factor between dA_i and A_j can be obtained by projecting A_j onto the surface of a **unit hemisphere** then projecting onto a **unit circle** on plane of dA_i, both centered at dA_i.

Hemicube ~Replace the unit hemisphere with half of a cube, i.e. $C = [-1,1] \times [-1,1] \times [0,1]$.
Analysis: Instead of double projection onto the plane, now there is one projection, onto the top or side faces of the hemicube.
Delta Form Factor - Amount of form factor contributed by a pixel on hemicube surface, if it is projected,

- Top $(x,y,1)$: $\triangle F_q = \frac{1}{\pi(x^2+y^2+1)^2} \triangle A$
- Side $(1,y,z)$: $\triangle F_q = \frac{x}{\pi(1+y^2+z^2)^2} \triangle A$, same for other 3

Implementation: 1) Consider a hemicube centred at patch i 2) Render and project all other patches onto the hemicube using hidden surface removal. 3) Find the pixels covered by each patch, and sum up their delta form factors.
Note: Based on **Item Buffering**, namely using colour buffer where each colour servers as a unique identifier.

Adaptive Subdivision

Subdivision Methods

(Naive) Adaptive Subdivision: 1) Use initial patches to compute a radiosity solution. 2) For every patch, subdivide if its local radiosity variation is large. 3) Recompute radiosity solution (from scratch) using all patches and sub-patches. 4) Repeat Step 2 & 3 until all local radiosity variations are within a threshold.
Analysis: Excessive re-computation: complexity $O((n+k)^2)$ for initial number of patches n new sub-patches k.

Sub-structuring Adaptive Subdivision ~(Improvement) During each subdivision of patch i, only compute form factors from each new sub-patch $i1, ..., ik$ to initial patches:

$$B_{iq} = E_i + R_i \sum_{j=1}^{n} B_j F_{(iq)j},\ q = 1, 2, ..., k$$

Analysis: [1] Complexity $O(nk)$. [2] Simple heuristic for hemicube: Subdivide if patch area is large compared to distance to nearest patches.

Progressive Refinement

Progressive Refinement ~A stream-based approach that continuously subdivides large (in terms of radiosity) patches until all small enough, addressing the issue of huge number of patches need to be first calculated requiring excess memory.
Procedure: 1) Find patch i with greatest unshot radiosity / emitted energy. 2) Compute form factors from patch i to all others. 3) Update the radiosity of each of the receiving patches. 4) If the new greatest unshot radiosity is below a threshold, stop the algorithm, otherwise repeat from Step 1.
Progressive Refinement updates patches using **Shooting** ~A single step commutes form factors from shooting patch to all receiving patches & distribute unshot energy $\triangle B_i$:

$$B_j^{(k+1)} = B_j^{(k)} + R_j F_{ji} \triangle B_i$$

Note: Update in B_j leads to unshot energy in B_j, for update in next iteration.

Others

Multi-Variable Calculus Misc

For a 3D surface $z = f(x,y), (x,y) \in D$ and $D \subseteq \mathbb{R}^2$, normal vector at surface point (x,y,z) is parallel to:

$$(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \pm 1)$$

Projected area of a plane with area s & normal N onto unit dir. Ψ:

$$s_{\text{proj}} = \cos\langle N, \Psi \rangle s = (N \cdot \Psi) s$$

From Past Year Papers

(AY19/20 Sem2 Midterm Qn2B) Vectors like `ecNormal` need to be normalised again in fragment shader even though the corresponding ones in vertex shader is normalsied, because normal vectors at vertices are bilinearly interpolated to the fragment, & resultings vector may not be a unit vector anymore.
(AY19/20 Sem2 Midterm Qn5B) Encoded perturbed normal vectors stored in a RGB image appears light blue, because unperturbed normal is $(0,0,1) \in [-1,1]^3$ and encoded to $(0.5, 0.5, 1) \in [0,1]^3$.
(AY19/20 Sem2 Midterm Qn6B) Mipmapping is not effective for shadow mapping boundary jaggies, as mipmapping only averages depth values while comparison results are still binary.
(AY19/20 Sem2 Final Qn29) If all patches in an enclosed space reflect all incoming light, the equilibrium radiosity will be **infinity** for all patches (& the linear system inconsistent).
(AY20/21 Sem2 Midterm QnD3) Differences between `texture()` and `texelFetch()`:

- Data - Former takes a normalised value in $[0,1]^2$ while latter uses exact int-valued texel coordinates.
- Retrieval - Former performs texture filtering based on the sampling parameters specified, while latter retrieves the exact one texel.

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.