Orthographic Camera Matrix

$$\begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{2}{near-far} & \frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

LookAt Matrix

$$\begin{bmatrix} r_x & r_y & r_z & 0 \\ u_x & u_y & u_z & 0 \\ d_x & d_y & d_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

r = right
u = up
d = direction

Positioning the Camera

$$\begin{bmatrix} r_x & r_y & r_z & 0 \\ u_x & u_y & u_z & 0 \\ d_x & d_y & d_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
lookAt(
    // the center of the "eye" (camera)
    eyeX, eyeY, eyeZ,
    // the center of our target
    centerX, centerY, centerZ,
    // the up vector to orient us
    upX, upY, upZ
)
```

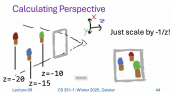
- To move forward?
  - Just update "center" towards the "eye" vector!
- To rotate?
  - Update your "eye" vector!
  - Requires a bit of cleverness
  - More detail to come!

```
aim:(aim_x,aim_y,aim_z,1)
aim_x = eye_x + cos(θ)
aim_y = eye_y + sin(θ)
aim_z = eye_z + Δtilt
```

Orthographic Camera Matrix

$$\begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{2}{near-far} & \frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Calculating Perspective



Z-Projection (Baseline)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Operations

First, scale **Z** by 2/(near-far)

Second, translate **Z** by 2/(near+far)

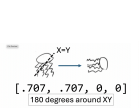
Third, apply projection

Fourth, translate **Z** back by -2/(near+far)

Multiplying the operations out

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$\alpha = \frac{near+far}{far-near} \qquad \beta = \frac{2near*far}{near-far}$$



×	1	i	j	k
1	1	i	j	k
i	i	−1	k	−j
j	j	−k	−1	i
k	k	j	−i	−1

- Any quaternion can be written as a *single* rotation matrix

https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation#Quaternion-derived_rotation_matrix

$$\mathbf{R} = \begin{bmatrix} 1-2s(q_j^2+q_k^2) & 2s(q_iq_j-q_kq_r) & 2s(q_iq_k+q_jq_r) \\ 2s(q_iq_j+q_kq_r) & 1-2s(q_i^2+q_k^2) & 2s(q_jq_k-q_iq_r) \\ 2s(q_iq_k-q_jq_r) & 2s(q_jq_k+q_iq_r) & 1-2s(q_i^2+q_j^2) \end{bmatrix}$$

Quaternion → "Axis Angle"

- Axis Angle representation
  - How we've informally been talking about geometry
- [x, y, z] axis plus θ
  - literally just a 4-tuple

angle = 2 * acos(qw)
x = qx / sqrt(1-qw*qw)
y = qy / sqrt(1-qw*qw)
z = qz / sqrt(1-qw*qw)

Quaternion → Euler Angles

- Not a unique transformation
- Many ways to define this, depends on your yaw-pitch-roll ordering

https://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles#Euler_angles_to_quaternion_conversion

$$roll = \alpha = \tan^{-1}\left(\frac{2(q_0q_1+q_2q_3)}{1-2(q_1^2+q_2^2)}\right) = \operatorname{atan2}(2(q_0q_1+q_2q_3),\ 1-2(q_1^2+q_2^2)) \quad (11a)$$

$$pitch = \gamma = \sin^{-1}(2(q_0q_2-q_3q_1)) \quad (11b)$$

$$yaw = \alpha = \tan^{-1}\left(\frac{2(q_0q_3+q_1q_2)}{1-2(q_2^2+q_3^2)}\right) = \operatorname{atan2}(2(q_0q_3+q_1q_2),\ 1-2(q_2^2+q_3^2)) \quad (11c)$$

Quaternions and SLERP

- SLERP turns out to be messy with Euler angles
  - Gimbal lock is a problem
- Quaternions turn out to be "better"
  - Computationally cheap
  - Decent for floating-point error

Quaternion exponents are defined on real numbers (q^t)

$q^t = \cos t\Omega + \mathbf{v} \sin t\Omega$

$slerp(q_0, q_1, t) = (q_1 q_0^{-1})^t q_0$

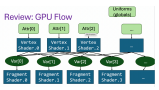
Review: Communication

- *Uniform*: constant data
  - resolution, transformations
- *Attribute*: describes a property for a **single vertex**
  - position, color, normal, ...
- *Varying*: **interpolated** by the **rasterizer** for all pixels

Primitive Assembly

- Connect the vertices into primitives
- *Only* calculates what vertices "make up" each primitive
  - Based entirely on vertex shader/attributes

Review: GPU Flow

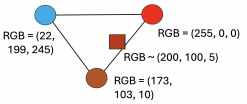


Window Space Transform

- Uses the `gl.Viewport` to calculate camera perspective for us!

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{width}{2}x_{ndc} + x + \frac{width}{2} \\ \frac{height}{2}y_{ndc} + y + \frac{height}{2} \\ \frac{farVal-nearVal}{2}z_{ndc} + \frac{farVal+nearVal}{2} \end{pmatrix}$$

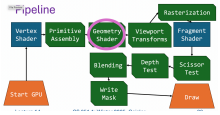
Review: Pretend to be a Rasterizer



Geometry Shader

- Apply operations to *primitives*
  - As opposed to vertices or fragments
- A bunch of (fairly niche) applications

Pipeline



Perspective Divide

- Homogeneous → Cartesian
- For each vertex:

`[x, y, z, w] → [x/w, y/w, z/w]`

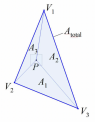
Rasterization

- https://en.wikibooks.org/wiki/GLSL_Programming/Rasterization <- extra details

$$\alpha_1 = \frac{A_1}{A_{total}} = \frac{\text{area of triangle } PV_2V_3}{\text{area of triangle } V_1V_2V_3}$$

$$\alpha_2 = \frac{A_2}{A_{total}} = \frac{\text{area of triangle } V_1PV_3}{\text{area of triangle } V_1V_2V_3}$$

$$\alpha_3 = \frac{A_3}{A_{total}} = \frac{\text{area of triangle } V_1V_2P}{\text{area of triangle } V_1V_2V_3}$$



Scissor Test

- Optional
- Cut out every fragment "outside" a specific window
- Box calculated based on the screen space
  - Calculates, for each fragment, draw if within the given (x, y) and width/height

Depth Test

- Optional
- Only draws a pixel if there is nothing "in front"
  - Defined by Z-value of each pixel
  - Maintained until the canvas is "cleared"

Blending

- "Blends" each color with a given attribute
  - Mapped by location relative to the draw position
- Extremely tricky to define exact behavior
  - Allows for partial transparency
  - Stronger than using "alpha", but much harder

Write Mask

- Optional
- Custom "masks"
- You can explicitly say "this pixel/color isn't drawn"
  - Can sharpen lines and apply color filters
  - Used for some fancy lighting stuff

Raytracing Idea

- All we need is light source + angle!
- So, just simulate a light source
  - Infinite points? We have finite fragments
  - Infinite rays? We have finite light sources + shapes

Raytracing Algorithm

Adapted from: https://lightcourses.stanford.edu/20/lectures/geometric/cpuarray/

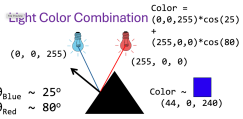
```
foreach pixel {
  ray = camera.cast(pixel)
  foreach triangle {
    if triangle.blocks(ray)
      maybe_bounce_or_reflect(ray)
    else if lightsource.intercepts(ray)
      pixel.light += light_source.color
  }
}
```

```
z_buffer = …
lights = …
foreach triangle {
  foreach pixel in triangle {
    if nearest_to_camera(pixel, z_buffer) {
      calculate_color(pixel, lights)
    }
  }
}
```

Lighting with Rasterization

- We don't want to simulate rays
- Instead, we directly estimate light contributions

Light Color Combination

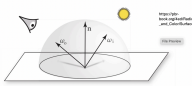


Rasterization

- Pixel-by-pixel triangle rendering
- What we've been doing!

BRDF

- **B**idirectional **R**eflectance **D**istribution **F**unction



Calculating BRDF

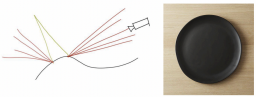
- Depends on how much "scatter" we give the light
  - How many points do we sample?
  - How much weight to give each angle?

Mirror BRDF

Only accept "perfect" reflectance

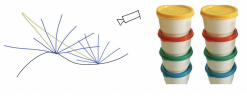
Glossy BRDF

Accept reflectance within a range



Diffuse BRDF

Every camera position gets the same amount of light



Why Inverse Transpose?

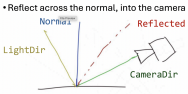
Consider a normal vector **N** and any surface vector **V**
We expect that **N**·**V**=0
Now, consider a transformation matrix **B** applied to **V**
We want to find a matrix **A** such that **AN**·**BV**=0
By definition of dot product, **AN**·**BV**=(**AN**)^T(**BV**)=0
After rearranging, we need $\mathbf{A}^T\mathbf{B}=\mathbf{id}$
Which implies that $\mathbf{A}=(\mathbf{B}^{-1})^T$

Diffuse Equation

```
vec3 lightDir = normalize(
    lightPos - fragPos
);
float diffuse = dot(
    lightDir, fragNormal
);
```

Specular Light

- Reflect across the normal, into the camera



Specular Code

```
vec3 lightDir = normalize(lightPos - position);
vec3 reflectDir = reflect(-lightDir, normal);
vec3 cameraDir = normalize(-cameraPos - position);
float angle = max(dot(cameraDir, reflectDir), 0.0);
// Account for "fall-off"
float specular = pow(angle, u_SpecPower);
```

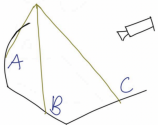
Putting it all Together

- Ambient Lighting
  - The "default light" in a scene\
- Diffuse Lighting
  - Light coming from a direction/source, but not focused
- Specular Highlight
  - Direct light from a source that bounces into the camera

Lighting Equation

```
vec3 color =
  u_AmbientLight +
    diffuse * u_BaseColor +
    specular * u_SpecColor;
```

Which BRDF "Pieces"?



Ambient: A, B, C
Diffuse: B, C
Specular: B

Gouraud Shading

- Let's just put all these components together
- Bonus performance: we can do all the calculations in the vertex shader
- Called "Gouraud Shading"

Attenuation Equation

```
float distance = length(light - pos);

float attenuation = constant
  + linear * distance
  + quadratic * distance * distance

vec3 color = (ambient + …) / attenuation
```

Spotlight Idea

- Just cut off any light "outside" of the spotlight
- Computed based on light direction
  - Which way the spotlight is facing matters

```
varying vec2 v_TexCoord;

uniform sampler2D u_Texture;

texture2D(u_Texture, v_TexCoord);
```

```
var texture = gl.createTexture();

gl.bindTexture(gl.TEXTURE_2D, texture);

gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA,
gl.RGBA, gl.UNSIGNED_BYTE, image);
```

What is Sampling?

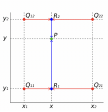
- We need to get data from somewhere
- Our texture is not always "aligned" with our size
  - Too small, or too big in our current dimension!
- We need to get information from "around" a single texel

Average Sampling Problems

- Average sampling actually is just bad
- We need to "weight" information, otherwise edges get weird and blurry

Bilinear Filtering

- Find the nearest 4 pixels
- Interpolate between them in two dimensions!
- We can extend this to get "trilinear" filtering



Swapping out Shaders

- Each GPU pass has its own uniforms/attributes
- We need to "swap out" *everything*
  - New program
  - New uniforms
  - New attribute pointers (!)

Framebuffer Operations

```
my_framebuffer = gl.createFramebuffer()

gl.bindFramebuffer(gl.FRAMEBUFFER,
    my_framebuffer)

gl.framebufferTexture2D(gl.FRAMEBUFFER,
  gl.COLOR_ATTACHMENT0, gl.TEXTURE_2D,
  my_texture, 0)
```

Framebuffer Summary

1. Setup a Framebuffer
2. Run the GPU to write to the Framebuffer
3. Swap the program/data to use the Framebuffer location as a texture

Reflection Idea

- We need to "see" from the perspective of the thing being reflected
- First, render the scene
- Second, put that render "on" the reflective shape

Practical Approach

- Capture the scene from angles all around the model
- Then sample that texture based on the normal of each vertex

"All around"

- In theory, we would want a "sphere" around the model
- In practice, there are too many angles
- Turns out that a cube (6 directions) is pretty good!

Reflection Steps

- Setup a Cubemap for the framebuffer
- Render the scene 6 times (one for each direction) with non-reflective models into the framebuffer
- Render non-reflective parts of your scene normally
- Render your reflective model

Shadowmap Smoothing

- We can use our texture filtering ideas
  - Bilinear, Anisotropic, Antialiasing
- A linear sampling is easy to implement
  - Done in-shader, not on the texture!

OpenGL Standardization

- Khronos was meant to help with standardization
  - Unify all of the commands into a common API
  - Lots of abstraction layers
- OpenGL was meant to be a "high level" API
  - Give the hardware design space
  - GLSL is similarly a "high level" language

Our Approach

1. 1st Pass: Encode the depth (Z) relative to the light for each fragment (rendered from the light)
2. 2nd Pass: Calculate the depth for each fragment relative to the light in the scene from the camera
3. 2nd Pass (cont): if these depths don't match, then the fragment is in shadow

Encoding the Depth

```
vec4 bitShift = vec4(256 * 256 * 256, …)
vec4 comp = fract(value * bitShift);
// apply mask
comp.w -= comp.z * 1.0 / 256.0;
comp.z -= comp.y * 1.0 / 256.0;
comp.y -= comp.x * 1.0 / 256.0;
return comp;
```

Projecting a Texture

- Just use a projection matrix from another direction!
  - The texture is 2D, so just sample from it
  - For shadows, we often want an orthographic projection

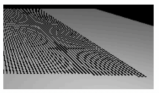
Decoding the Depth

```
float total = 0.0;
total += color.r / (256.0*256.0*256.0);
total += color.g / (256.0*256.0);
total += color.b / (256.0);
total += color.a;
return total;
```

Putting it together

- We treat shadows as "perfect"
- Something either has light (normal lighting) or is just a perfectly (0, 0, 0) pixel

Fixing the Problems



Add a floating-point / z-fighting correction term

Disable CULL_FACE

Change the Light Ortho Projection

HLSL

DirectX

- "High Level Shading Language"
  - Very GLSL-like

```
PSInput VSMain(float4 position : POSITION, float4 color
: COLOR) {
    PSInput result;
    result.position = position;
    result.color = color;
    return result;
}
```

Metal Shader Language

```
fragment float4 textureFragmentShader(
  TexturePipelineRasterizerData in,
  texture2d<float> texture) {
    sampler simpleSampler;
    float4 colorSample =
texture.sample(simpleSampler, in.texcoord);
    return colorSample;
}
```

Vulkan

Vulkan

- The "low level" API
  - Been called "assembly for the GPU"
  - Way more control over data movement
  - Extra control over memory and thread scheduling
- Has become quite popular recently
  - Good for game dev especially

The Future? Slang

Slang

- Adds modern language features
  - Generics/Interfaces and nice type systems
  - Fancy Tooling and cross-compilation
- Focus on managing shader pipelines
  - Decompose and optimize

What is a "Compute Shader"?

- Originally, GPGPUs just needed to read textures
  - Computation was sort of "tacked on"
- Now, there's a 'separate' set of compute stages
  - Still use the hardware of the pipeline
  - But "streamline" the computational focus

OpenMP

OpenMP

- Old, classic, high-performance computing framework
  - Extension to C, C++, FORTRAN
- Can be used for compute shaders nowadays

CUDA

NVIDIA CUDA

- NVIDIA-specific language
  - Built to look like (and work with) C and C++
  - It is a fully distinct language (like GLSL)
- Focused on thread management (like OpenMP)
  - More emphasis on making distinct "shader/kernel"

OpenCL

OpenCL

- Managed by Khronos
- The "default" GPGPU API besides CUDA
  - Supported by most (all?) non-NVIDIA GPUs
- Also supports more general heterogeneous programming
  - FPGAs and other hardwares

Summary + Questions

- Graphics APIs
  - OpenGL, DirectX, Metal, Vulkan, Slang
- GPGPU (Compute) APIs
  - OpenMP, CUDA, OpenCL
- Graphics-related compilation
  - SPIR-V

Search Labs | AI Overview

A frame buffer is a section of memory that holds the data for an image that appears on a computer screen. It's also known as a framestore.

How it works

- The graphics engine updates the frame buffer with the next image to display
- The frame buffer circuitry in a video card converts the bitmap into a video signal
- The video signal is then displayed on a computer monitor

Frame buffer components

- **Color buffer**: Holds the color data for an image
- **Depth buffer**: Holds a depth value for each pixel
- **Stencil buffer**: Holds additional information for creative rendering

Frame buffer in WebGL

- A framebuffer in WebGL is a data structure that organizes the memory resources needed to render an image
- Additional framebuffers can be created for off-screen rendering

Frame buffer location

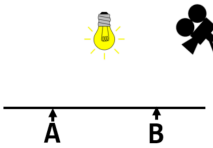
- The frame buffer can be a separate memory bank on the graphics card, GPU, or a reserved part of regular memory

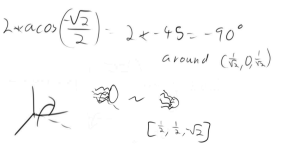
We loosely defined depth to be "further into the clip space box", and is thus proportional to distance. Since our z component gives us distance in a perspective projection (and arguably in an orthographic projection), we can directly just use this number. Given a depth of a fragment relative the camera, we can determine if there is *something* with smaller depth, which in turn tells us for each fragment if it's in shadow or in light.

A perspective camera scales each object proportional to the distance from the camera (where size ~ 1/z). An orthographic camera, in contrast, only cuts off objects if they fall outside of the viewing plane, but does not scale more-distant objects.

At point A, we would likely observe Ambient and Diffuse components of light, but not specular. Ambient light is visible to any fragment visible to the camera, and Diffuse light is visible (at least somewhat) to any surface visible to the camera and facing vaguely towards the light. The specular component, however, is likely not visible, since the bounce of the specular will shoot to the left rather than towards the camera.

At point B, we would likely observe Ambient and Diffuse components of the light, for reasons identical to point A. We would, however, likely also observe some amount of the specular light component, since the specular "bounce" seems to line up pretty exactly with the camera position.





→ color = (W * (.3 - .2) + B * (.2 - .1)) / (.3 - .1) ~ ((25, 25, 25) + [0, 0, 25]) / .2 ~ (25, 25, 50) / .2 ~ (125, 125, 250)