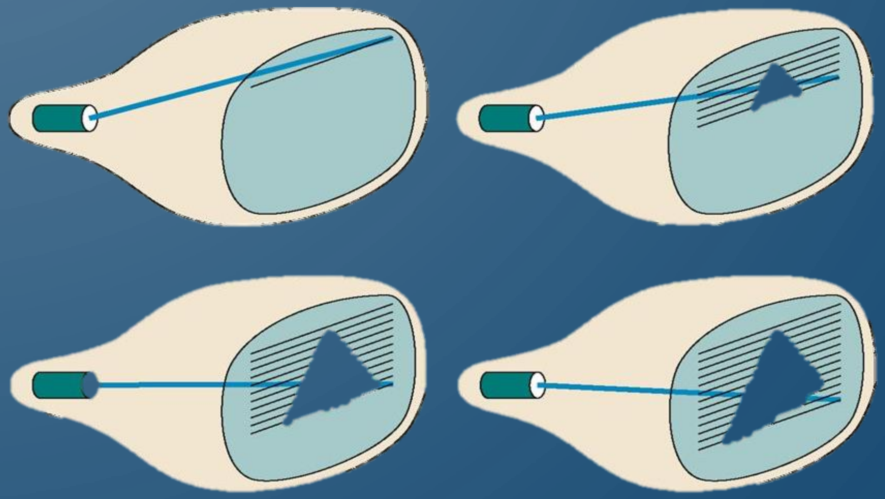
GPAT – CHAPTER 2, 4, AND 8
GRAPHICS AND CAMERAS

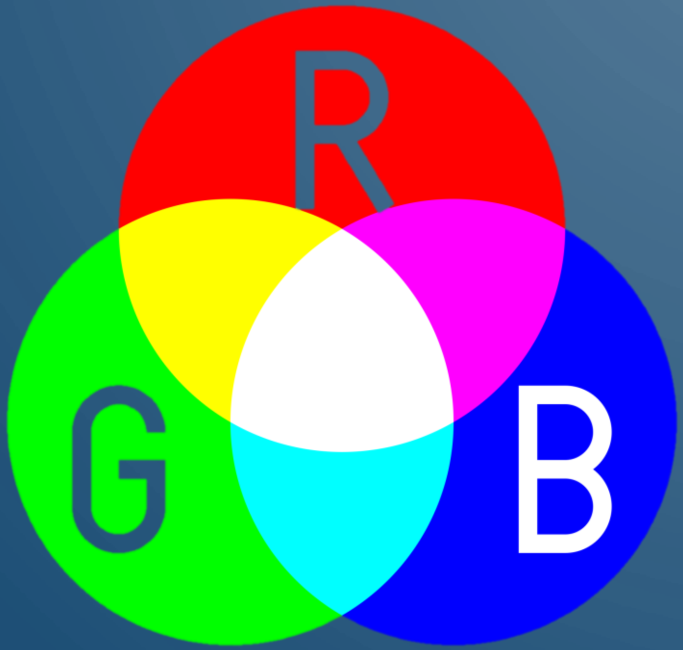
SOME BASICS OF GRAPHICS

- A **pixel** is a picture element whose data is typically at least a color (but can be more, e.g., depth information)
- The **framebuffer** is special location in memory of pixel data for the monitor to display
- Monitor technology (e.g., CRT) used to be built upon the concept of a **scan line**, i.e., a row of pixels, and many algorithms still rely on this.



Cathode Ray Tube Monitor

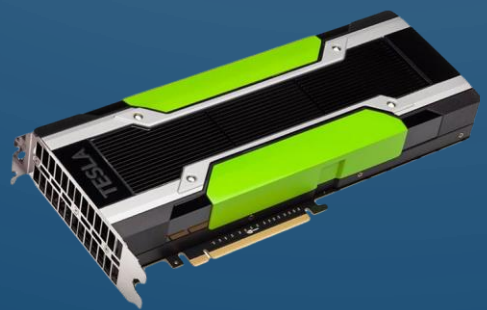
SOME BASICS OF GRAPHICS



- Colors are usually expressed in Red-Green-Blue (RGB) format
  - 3 8-bit integers (each a value 0-255) or 3 floating-point numbers (each a value 0-1) representing intensity. 0 is no intensity or black
- Often colors add in an alpha channel representing transparency. 0 is fully transparent. 255u or 1.f is fully opaque.

SOME BASICS OF GRAPHICS

- Modern computers and consoles have **graphics processing units (GPUs)**
  - Knows how to render points, lines, and triangles
  - Has dedicated memory
  - Executes **shaders**, or small programs, to operate on data
  - Operates on 4-byte floating point numbers



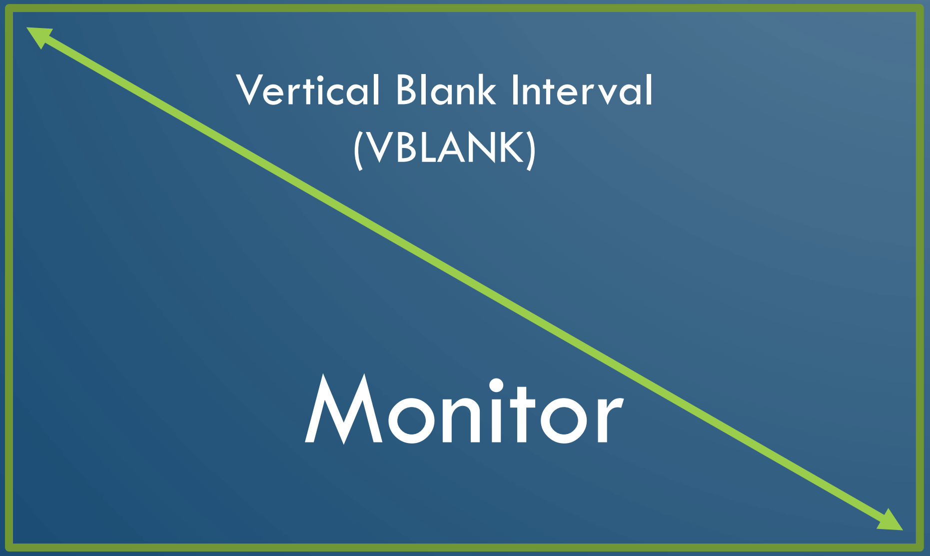
- Things to keep in mind:
  - Geometry data lives on GPU, not CPU – data transfer occurs through memory **buffers**
  - Picture information, often called a **texture**, also lives in GPU memory – also called a color map
  - GPUs are highly parallel, CPUs are not
  - GPUs have limited memory that must be managed properly

DOUBLE BUFFERING

THERE IS A PROBLEM THOUGH!

- What happens when the CPU changes the framebuffer while the monitor is drawing it?
- **Screen tearing** or showing two partial frames at once

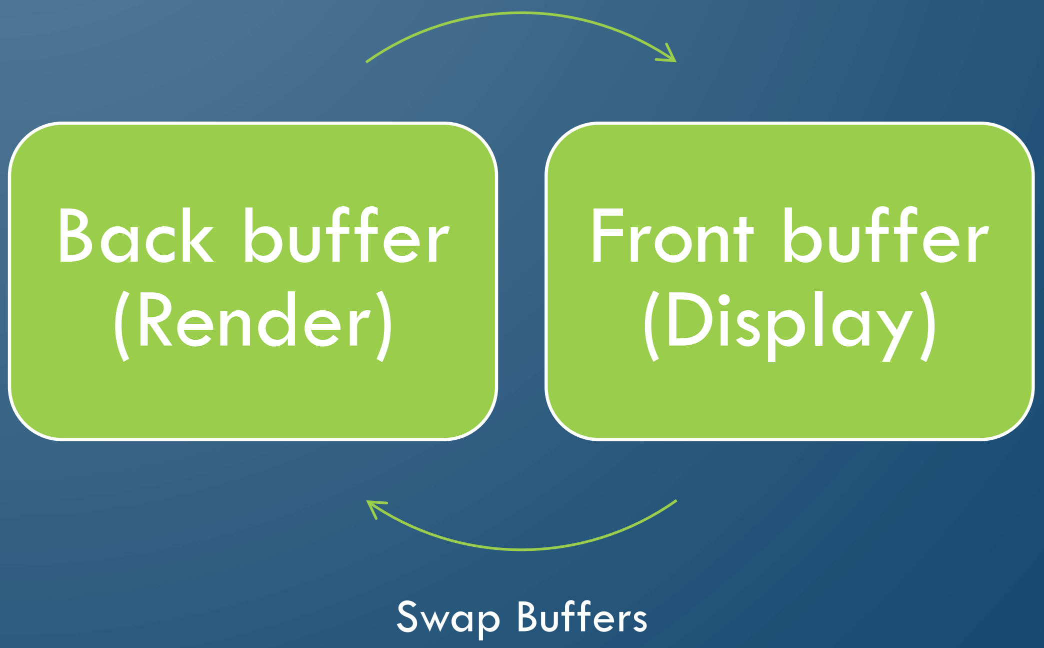
SOLUTION – VSYNC



- Synchronize the game loop rendering with the interval that the scanner is returning to the original position, called vertical blank interval
- Problem?
  - Still not enough time!

SOLUTION – DOUBLE BUFFERING

- Have two frame buffers
  - Render to the back buffer
  - Display the front buffer
  - At the end of the game loop rendering
    - `Wait for VBLANK`
    - `Swap frame buffers`



SPRITES

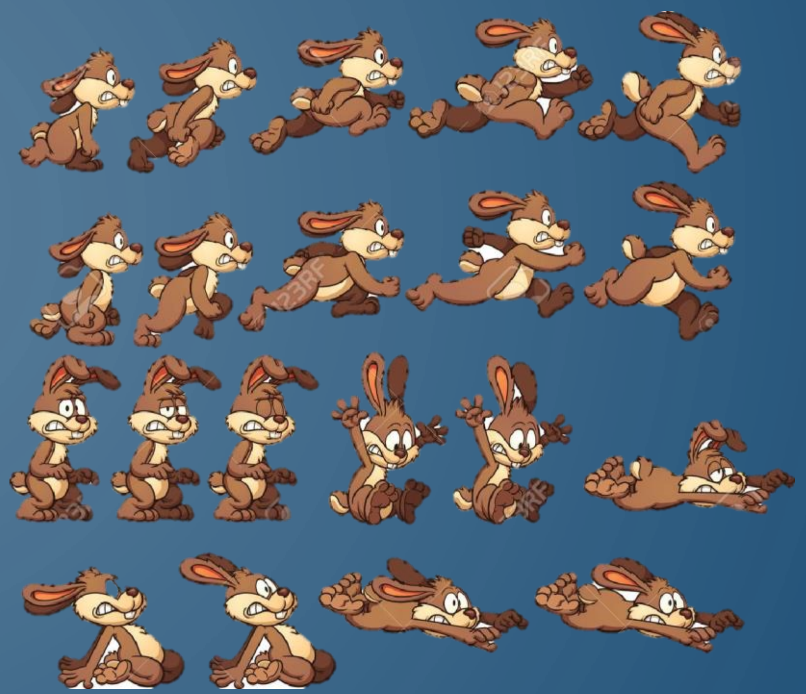
DRAWING SPRITES

- A **sprite** is a 2D visual game object that can be drawn with a single image
  - Examples – characters, objects, backgrounds
- 2D games have dozens to hundreds of sprites to manage as texture objects (its these game assets that make them large)
- How should we draw them?
- Draw in order of background to foreground, called the **painter's algorithm**
  - Give each sprite an integral "draw order"
  - Some libraries break further into layers, and each layer is drawn based on draw order

- How do you store all of the sprites?
  - Sorted container
  - Update step should set draw order and re-sort container

```
class Sprite {
  ImageFile image
  int drawOrder
  int x, y
  void draw() {
    // Draw image at correct
    // (x, y)
  }
}
```

ANIMATING SPRITES



- Based on "flipbook" animation
  - Show series of images fast enough
- Store an array of sprite images in order of animation

```
struct AnimFrameData {
  int startFrame; // Starting index for animation
  int numFrames;  // Number of frames in animation
}
struct AnimData {
  ImageFile images[];          // All sprites for animations
  AnimFrameData frameInfo[]; // Animation information
}
```

ANIMATING SPRITES

```
class AnimSprite extends Sprite {
  AnimData animData; // All animation data
  int animNum;       // Active animation
  int frameNum;      // Frame of active animation
  float frameTime;   // Amount of time current frame has been displayed
  float animFPS;     // FPS of animation

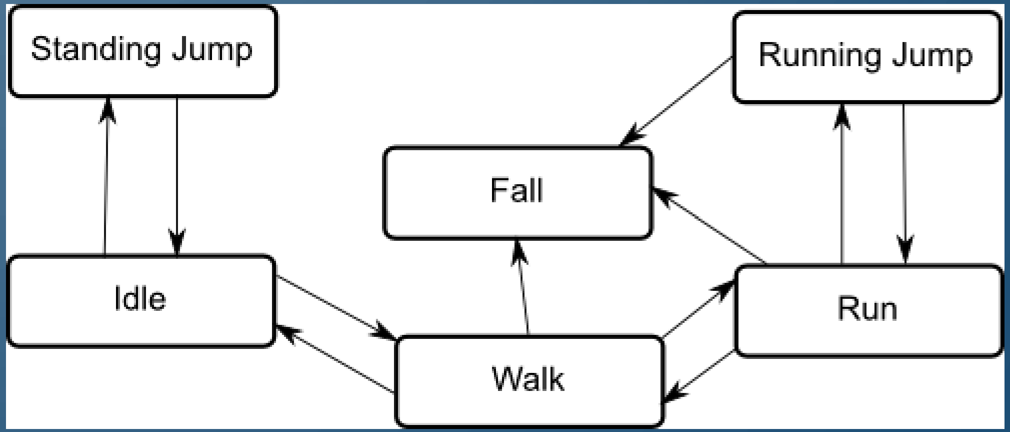
  void initialize();                    // Create/set animData and
                                        // starting animation
  void updateAnim(float deltaTime);     // Update based on delta game time
  void changeAnim(int num);             // Resets frameNum and frameTime to
                                        // 0 and sets image to first of
                                        // animation num
}
```

ANIMATING SPRITES

```
void updateAnim(float deltaTime) {
  frameTime += deltaTime;
  // Check to advance to next animation frame
  if(framTime > 1/animFPS) {
    // Advance (frameTime / (1/animFPS)) frames
    frameNum += frameTime * animFPS;
    // Wrap animation
    frameNum %= animData.frameInfo[animNum].numFrames;
    // Update image and frameTime
    int imageNum = animData.frameInfo[animNum].startFrame + frameNum;
    image = animData.images[imageNum]
    frameTime %= 1/animFPS;
  }
}
```

HOW DO YOU SWITCH BETWEEN ANIMATIONS?

- Use a state machine
  - More on this when covering AI
- Essentially, a graph
  - Nodes are specific animations (pick one to start on)
  - Edges represent transitions
    - Automatic (e.g., after 3 seconds)
    - Action (e.g., after pushing 'A')



SPRITE SHEETS

- Efficient file representation for sprites. Put them all in a single texture (packed closely)

SCROLLING

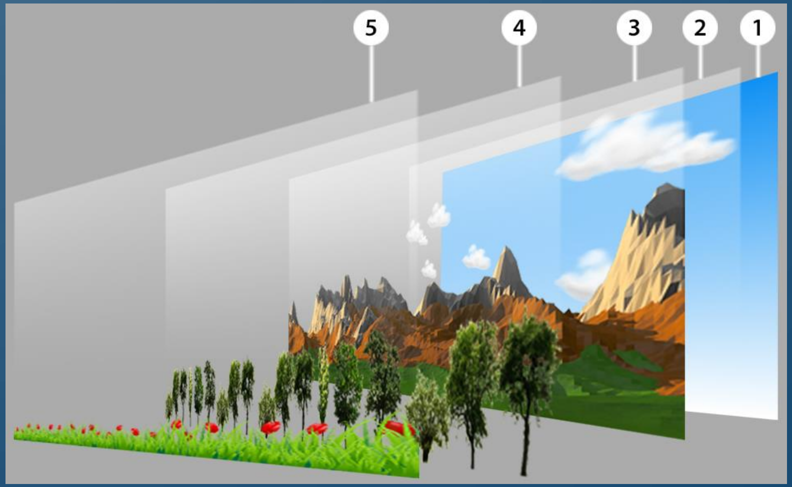
SINGLE-AXIS SCROLLING

- Assume we have a finite set of images, all screen-sized segments (e.g., 960x640) scrolling on x-axis
- Initialize ith image x at imageIndex*screenWidth
  - 1st image at 0, 2nd at 960, 3rd at 1920, …
- How many backgrounds should be drawn at a time?
  - 2
- Need x, y coordinates of "camera"
  - Starts at center of first screen
  - Lets have camera x be the players x, except cannot go behind first image/past last image

SINGLE-AXIS SCROLLING

```
camera.x = clamp(player.x, screenWidth/2,
    imageCount * screenWidth - screenWidth/2);

Find image i camera is in by camera.x/screenWidth;
Draw image i at (i.x - camera.x + screenWidth/2, 0);
Draw image (i+1) at (i.x - camera.x + screenWidth/2, 0);
```



SCROLLING EXTRAS

- **Infinite scrolling** can be implemented by looping through images (wrapping) or randomly piecing image sequences together
- **Parallax scrolling** breaks background into multiple layers at different depths
  - Typically need at least 3 layers
  - Implemented by `drawing image i at (i.x – (camera.x – screenWidth/2) * speedFactor, 0)`
    - Note need different find equation
- **Four-way scrolling**
  - Incorporate the y-axis too
  - Have matrix of background images
  - How many images should be drawn?
    - 4

TILE MAPS

CREATING WORLDS WITH TILE MAPS

- **Tile maps** are a partitioning of the world into polygons of equal size (e.g., squares, parallelograms, or hexagons)
  - Each tile represents a sprite as a numeric lookup into the **tile set**

SIMPLE TILE MAPS (GRID)

- Step 1: determine size of tiles
- Step 2: think of a file format to design tile maps
  - 5,5
    0 0 1 0 0
    0 1 1 1 0
    1 1 2 1 1
    0 1 1 1 0
    0 0 1 0 0
- Step 3: class representation
  - ```
    class Level {
      const int tileSize = 32;
      int width, height;
      int tiles[][];
      void draw() {
        for(int[] row : tiles)
          for(int tile : row)
            // Draw tile at
            // (col*tileSize, row*tileSize)
      }
    }
    ```
    ```

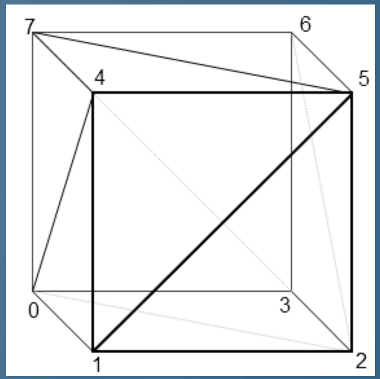
ISOMETRIC TILE MAPS

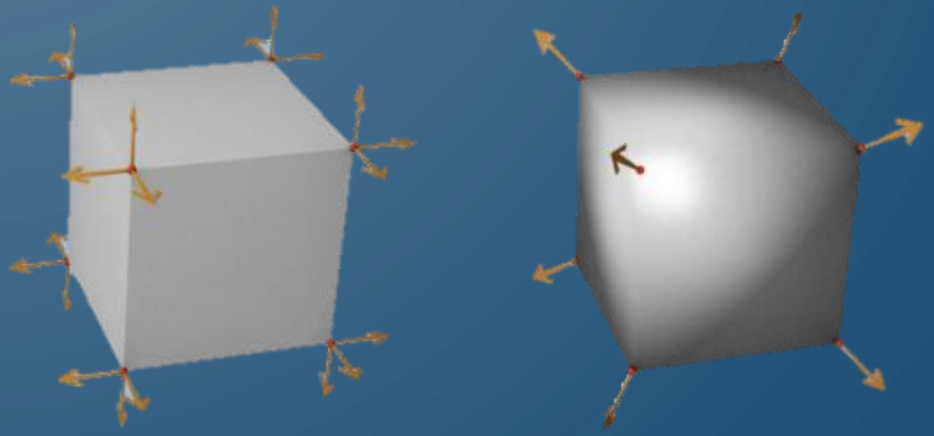
- Use diamonds or hexagons
- Can utilize multiple layers
  - Higher levels have more complex/larger structures
- Complex, but you can definitely figure them out! Get creative!

3D VIEWING PIPELINE

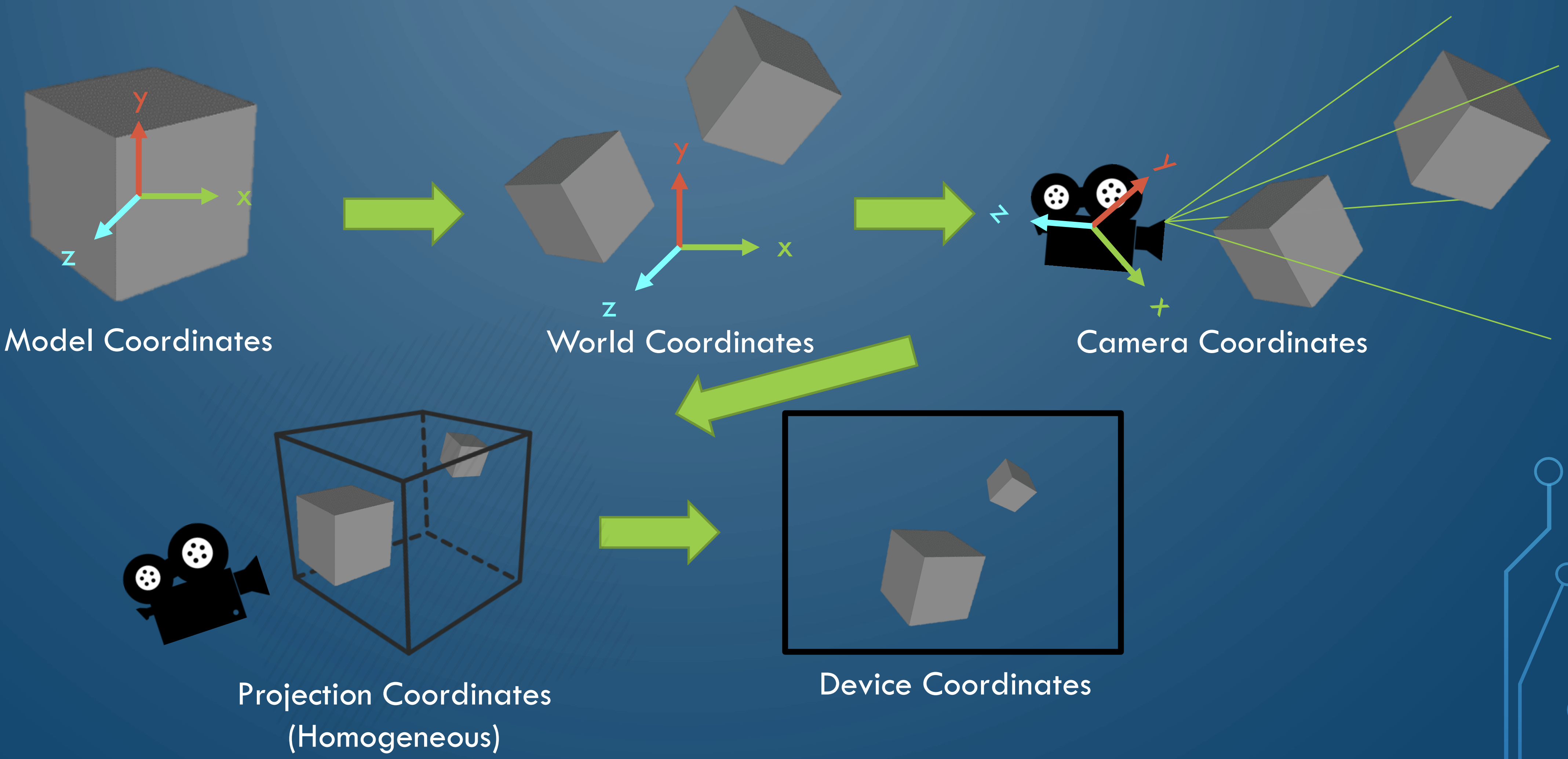
DEFINING MODELS

- Models are polygonal **meshes**
  - Vertex data
    - Position
    - Normal
    - Texture coordinate
    - Etc
  - Face data (triangles)

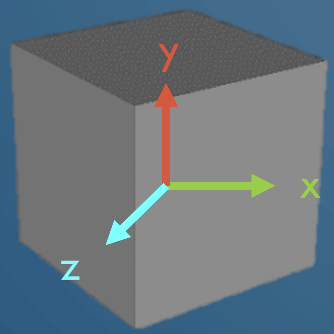




VIEWING PIPELINE



MODEL SPACE

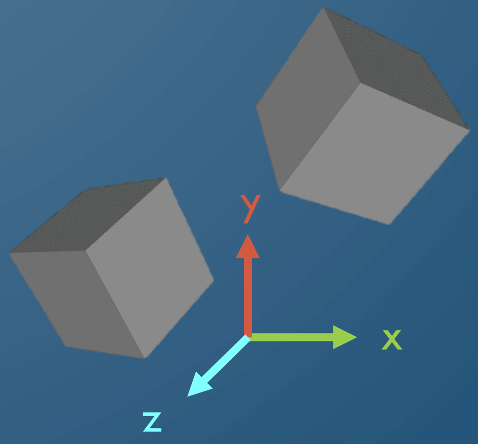


Model Coordinates

- Origin is typically the center of mass of the object, or a vertex
  - Humanoids might have the origin at the feet

WORLD SPACE

- Origin is a special point in the space
- Models are transformed into this virtual scene
  - Scaled
  - Rotated
  - Translated
- Homogeneous coordinates – use 4D vectors with the 4th component usually 0 (direction) or 1 (point)



World Coordinates

WORLD SPACE

- Points get transformed by a series of matrix manipulations

$$p' \leftarrow Mp$$

- Translation

$$M = T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Scale

$$M = S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Rotation

$$M = R_x(\theta)$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$M = R_y(\theta)$$

$$M = R_z(\theta)$$

WORD SPACE

- Homogeneous coordinates allow for translation to be a matrix transformation
- Imagine the various transforms to see how they work, for example

$$p' \leftarrow T(t_x, t_y, t_z)p = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

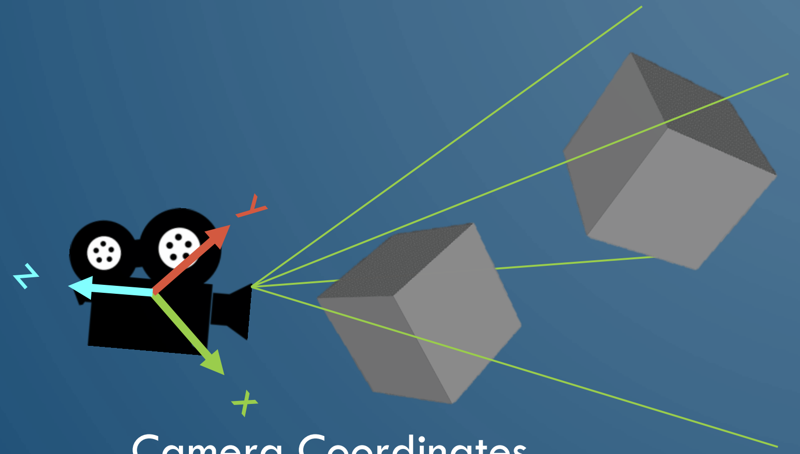
$$=?$$

- To apply multiple transformations

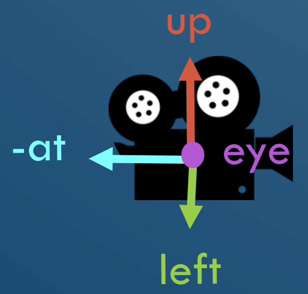
$$M = W = TR_zR_yR_xS$$

- Why is rotation and scale performed before translation?

CAMERA SPACE



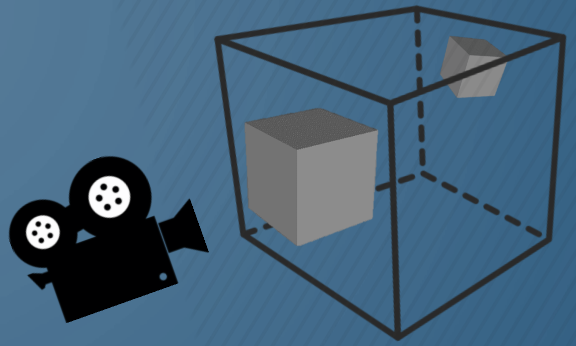
Camera Coordinates



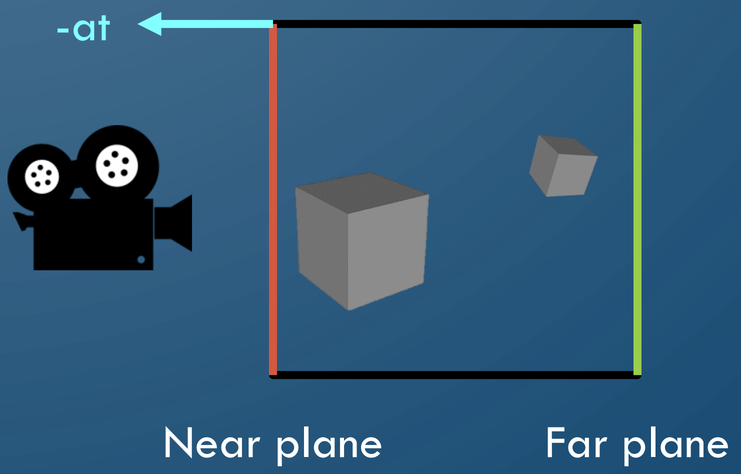
- Origin is now the camera
- Axes are defined by the direction the camera faces
- Camera definition
  - Eye – world position of camera
  - At – unit vector of camera $-z$-axis
  - Up – unit vector of camera y-axis
- Transformation matrix computed an applied to all objects (look-at matrix)

PROJECTION SPACE

- Many projection options (always converts scene to homogeneous cube)
  - **Orthographic projection** – parallel lines stay parallel and object size is not relative to distance from camera
  - **Perspective projection** – parallel lines converge and object size is relative to distance from camera
    - Defined by **field of view** and aspect ratio

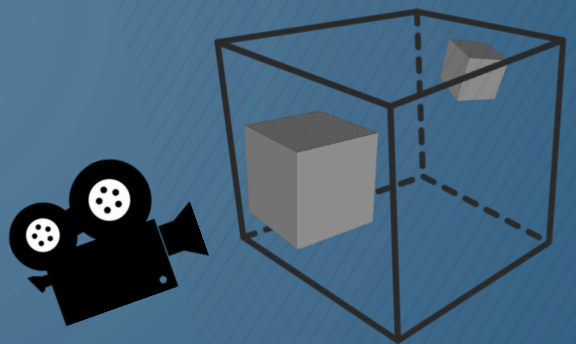


Projection Coordinates (Homogeneous)

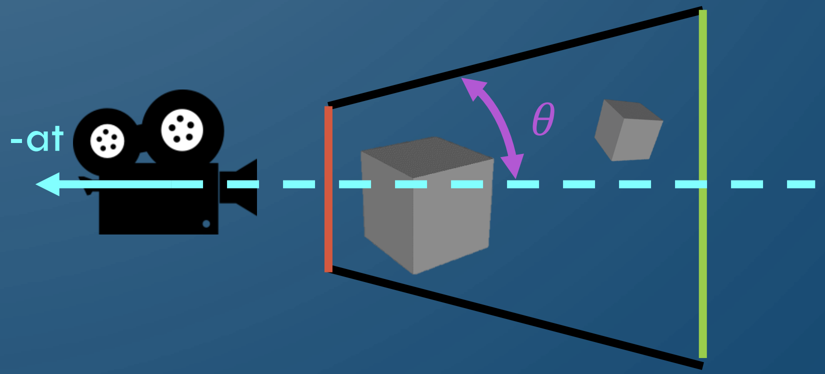


PROJECTION SPACE

- Objects outside of homogeneous cube are clipped from the scene (performed efficiently on GPU)
  - **Near plane** is closest visible z-coordinate to camera
  - **Far plane** is farthest visible z-coordinate to camera
- Again transformation performed by a transformation matrix

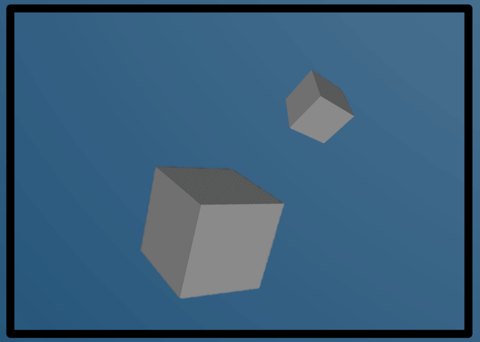


Projection Coordinates (Homogeneous)



Near plane Far plane

DEVICE SPACE



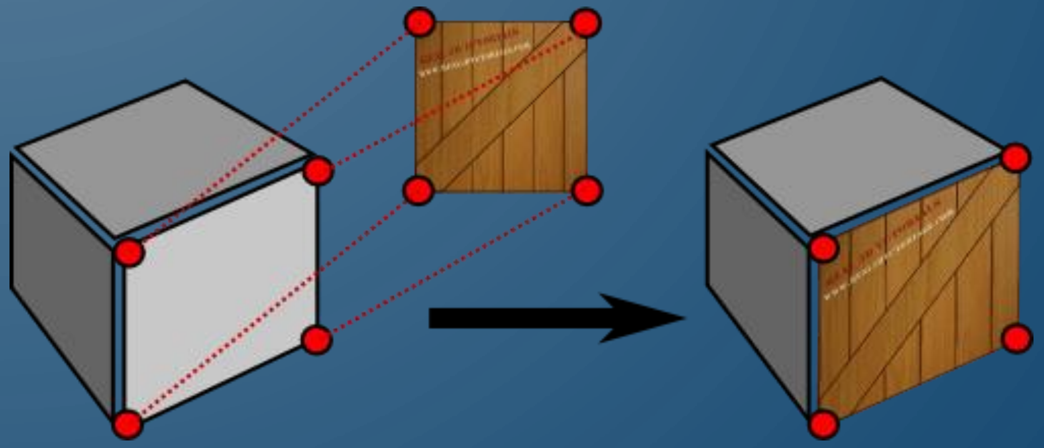
Device Coordinates

- Coordinates are transformed (by another matrix computation) to **viewport** coordinates (essentially x, y screen positions)
- For efficiency, as many matrices as possible are multiplied together before being applied to objects
  - Why?
  - Which matrices can be collapsed?

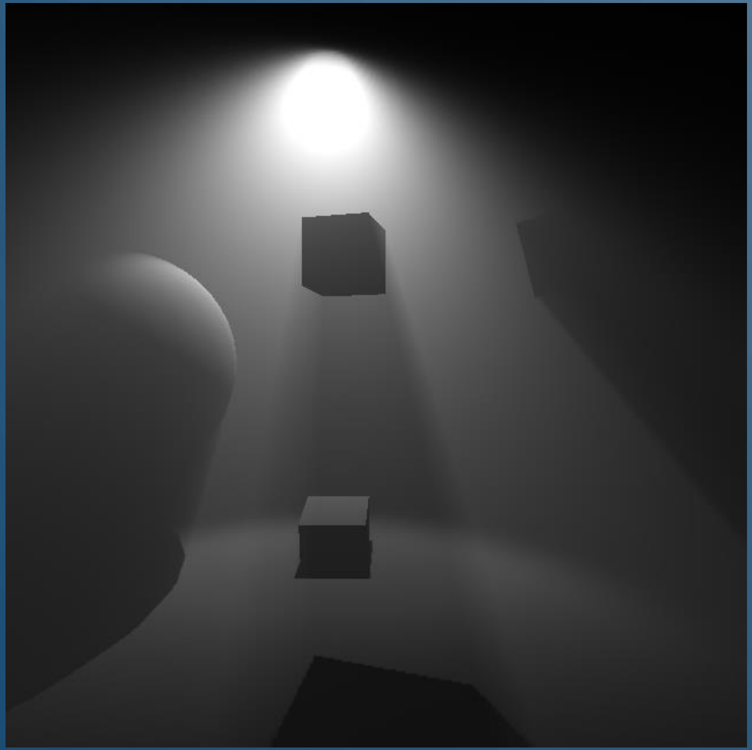
LIGHTING

TEXTURE MAPPING (NOT LIGHTING)

- Gives an object its base color
- Each vertex has a **texture coordinate** which refers to a location in a texture
  - Texture coordinates are always in $[0,1]^2$ and are not pixel coordinates
  - Also called UV coordinates



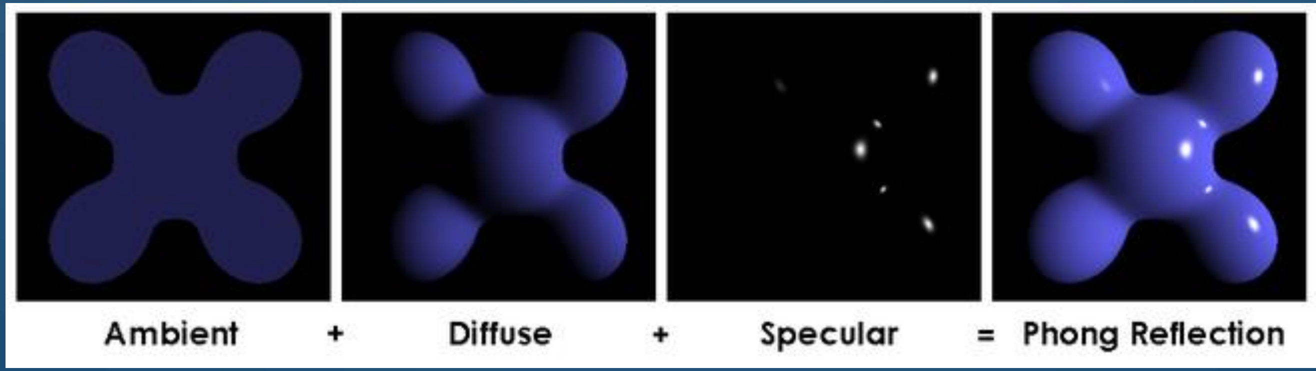
LIGHTS



- Many types of lights, we will look at most basic ones
  - **Ambient light** – uniform amount of lighting in a space
  - **Directional light** – light without a position that affects entire scene, e.g., sun (single directional light per scene usually)
  - **Point light** – light with a position emitting light in all directions, e.g., lightbulb
  - **Spotlight** – light with position and direction, e.g., flashlight

PHONG REFLECTION MODEL

- Local lighting model – no secondary light reflections, i.e., object lighting is not affected by other objects
  - **Ambient light** – base illumination from scene
  - **Diffuse light** – primary reflection of light that is evenly scattered
  - **Specular light** – shiny reflections of light based on viewing direction



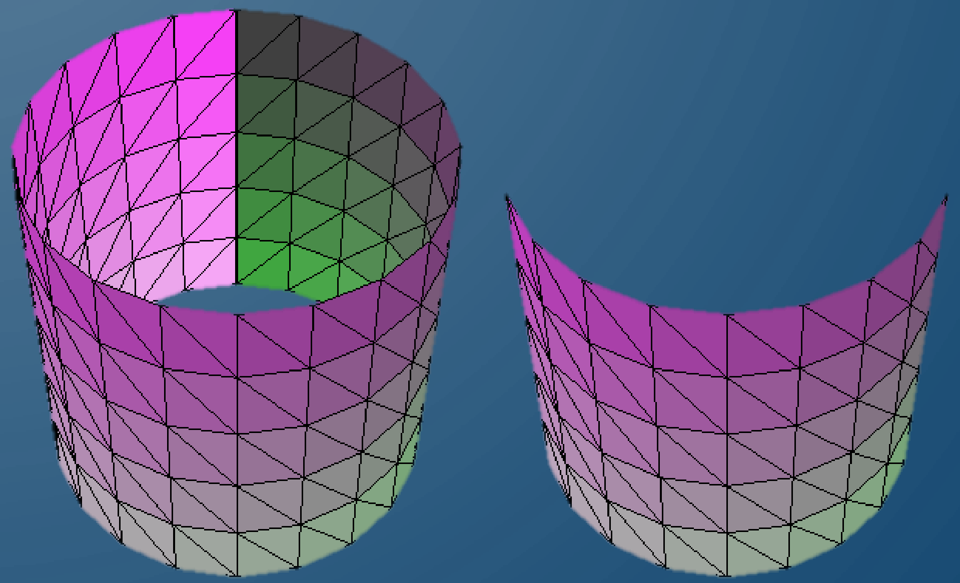
SHADING

- Shading is the determination of how the surface of a triangle is filled in, with respect to the lighting model
  - **Flat shading** – Uses face normal to compute light model one time and applies that color uniformly
  - **Gouraud shading** – light model computed for each vertex and color is interpolated
  - **Phong shading** – Vertex normal interpolated and light model computed for every pixel

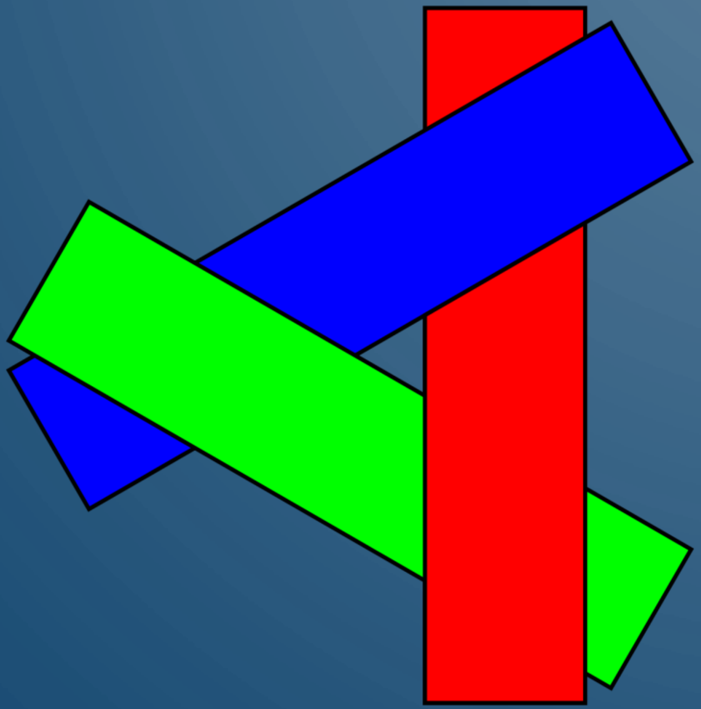
VISIBILITY

BACK-FACE CULLING

- Remove triangles from rendering which do not face the camera
- Performed by analyzing dot product of face normal with camera at vector, if negative then do not render



PAINTER'S ALGORITHM (AGAIN)



- Draw items in background to foreground
- Any issues?
  - Order ill specified
  - Required resorting each frame
  - Overdraw (recomputing pixel color over and over again)

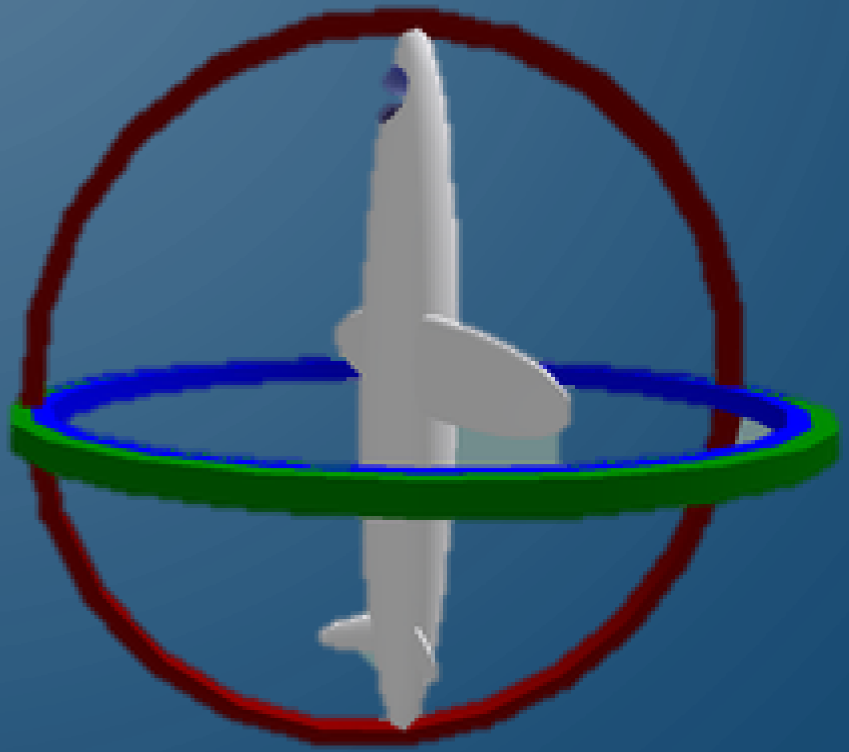
Z-BUFFERING

- The **z-buffer** is additional memory (in frame buffer) that stores depth (distance from camera) information of pixel
- During rendering, we only update a pixel's color if a pixel is closer than currently stored in the z-buffer
- Any issues?
  - Floating-point error
  - Transparency?
- To handle transparency
  - Draw all opaque objects
  - Make z-buffer read only
  - Draw all transparent objects
- Note – professional game engines employ many more techniques for efficient visibility determination

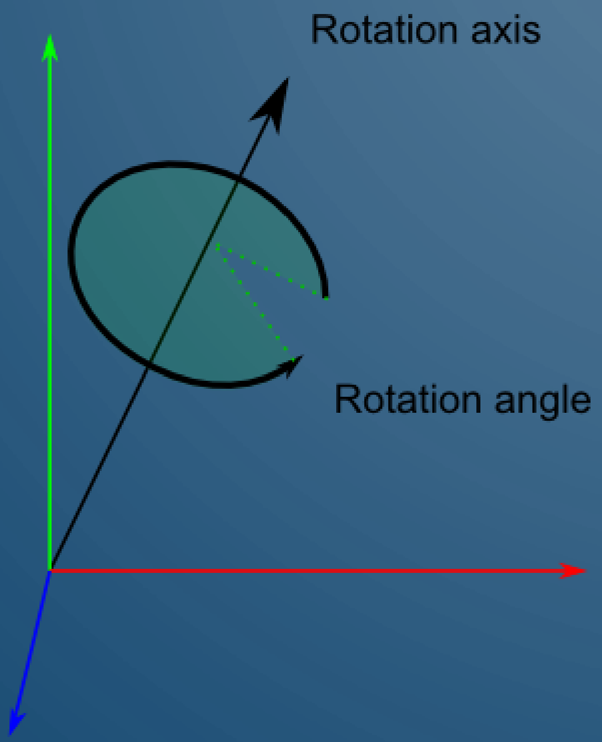
WORLD TRANSFORM, REVISITED

REPRESENTING ROTATIONS

- Euler angles
  - 3 separate angles (essentially as discussed)
  - Difficult to interpolate
  - Gimbal lock
- Rotation matrix
  - 16 values
  - Expensive to interpolate



REPRESENTING ROTATIONS



- Angle-axis
  - More intuitive
  - Store an axis of rotation and angle of rotation
  - Difficult to interpolate as is

REPRESENTING ROTATIONS

- Quaternions
  - Alternative representation of angle-axis
  - Small storage – 4 values
  - Smooth interpolation
  - No gimbal lock

- What could be terrible about them?
  - Most confusing mathematical concept you may ever learn (unless you jump into higher level math)
  - Unintuitive!
  - Tradeoff – will always need to convert to rotation matrix to actually transform object (but not really a negative)

QUATERNIONS

- "A 3D complex (real + imaginary) number"
- Useful in representing 3D rotations, essentially, angle-axis rotations
- Representation
  - Scalar value
  - Vector component (imaginary component)
- In graphics, we will always have unit quaternions (magnitude of 1)

- $q = \langle q_s, \overrightarrow{q_v} \rangle$
- From angle-axis $(\theta, \hat{a})$
  - $q = \left\langle \cos\frac{\theta}{2}, \hat{a}\sin\frac{\theta}{2} \right\rangle$
- Libraries often provide convenient construction mechanisms from Euler Angles or Angle-axis rotations
- Mathematics has many useful operations combined, e.g., multiplying (combines rotations), conjugation (inverse), etc.
- Quaternion rotation applied to a point

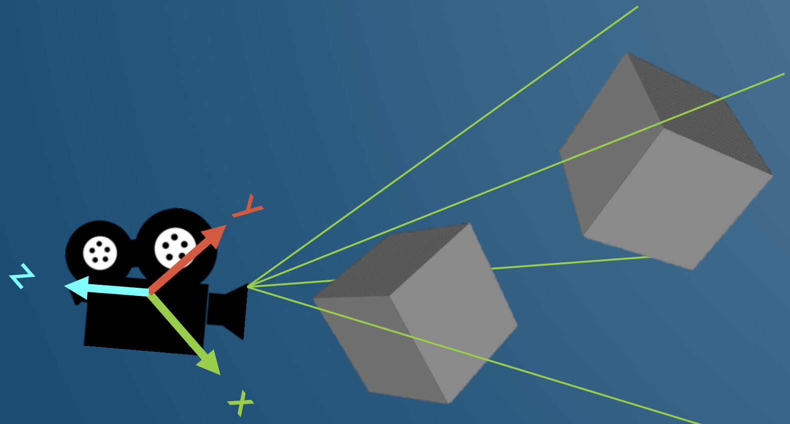
$$p' = q^{-1}pq$$

CAMERA MODELS

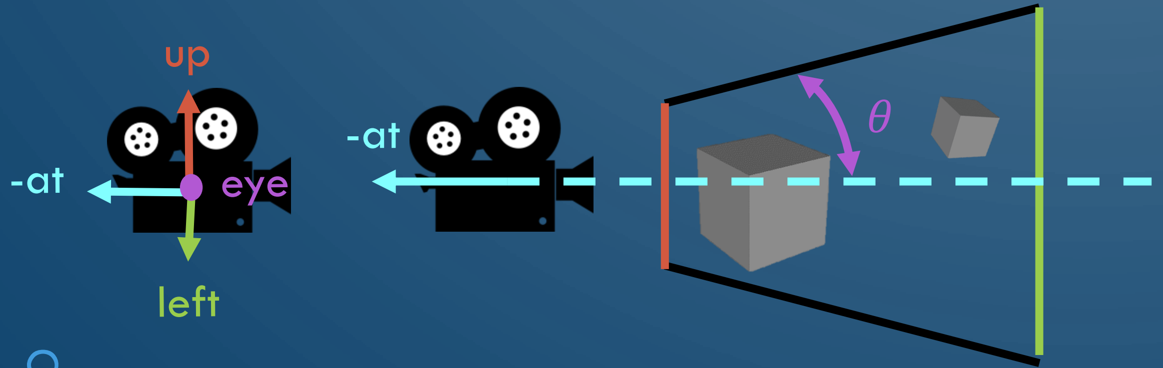
TYPES OF CAMERAS

- **Fixed and non-player controlled cameras** – same position or scripted positions by designer
- **First-person camera** – gives perspective of the player
  - Need to worry about player model used in rendering
- **Third-person camera** – possibly an omniscient perspective of the world
- **Follow camera** – limited view that follows player in world
- **Cutscene camera** – designed with smooth transitions using spline system

REVIEW OF CAMERAS AND PERSPECTIVE



Camera Coordinates

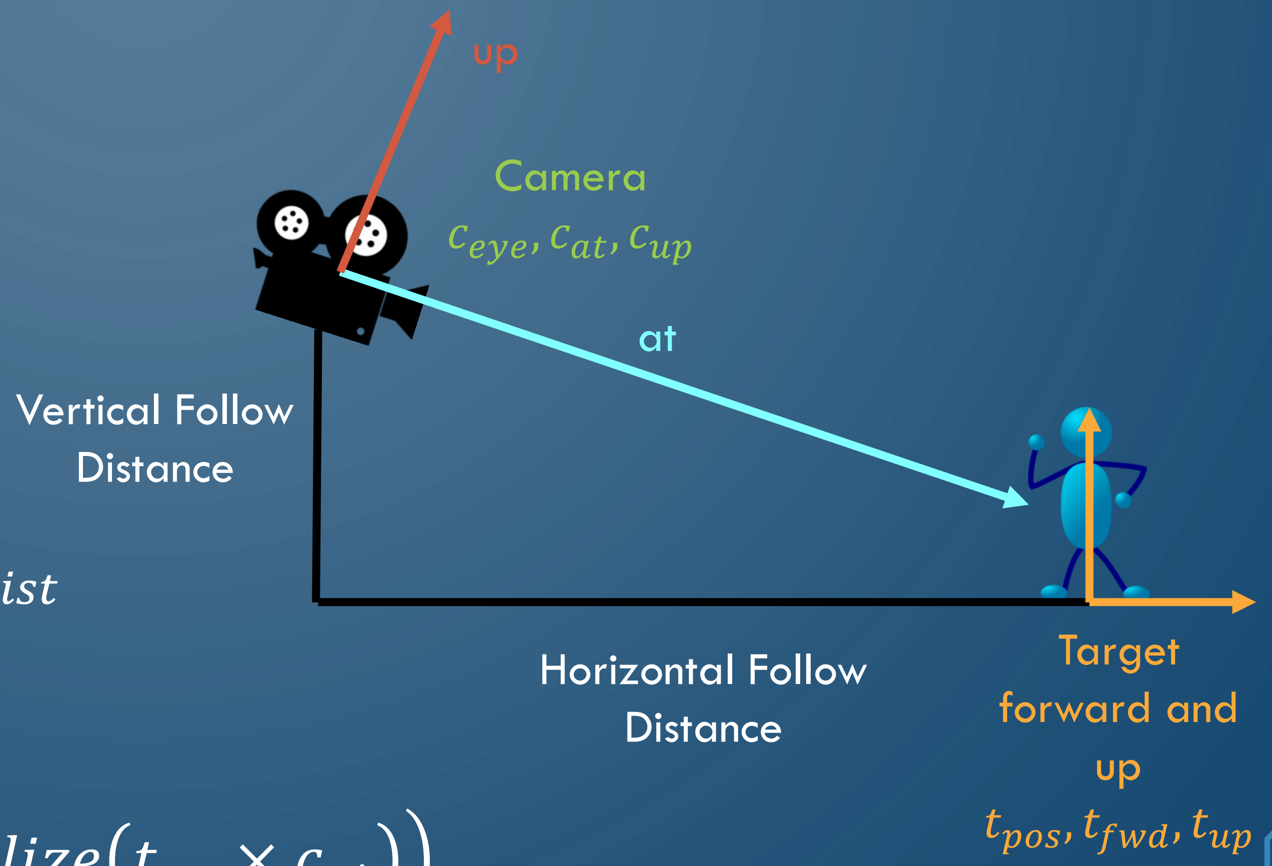


Near plane Far plane

- Cameras defined by eye position, look-at direction, and up direction
- **Perspective projection** defined by **field of view (FOV)**, **aspect ratio**, near plane and far plane
  - Careful of the **fisheye effect** when the FOV is too large

BASIC FOLLOW CAMERA

- $c_{eye} = t_{pos} - t_{fwd} h_{dist} + t_{up} v_{dist}$
- $c_{at} = normalize(t_{pos} - c_{eye})$
- $c_{up} = normalize\left(c_{at} \times normalize(t_{up} \times c_{at})\right)$



SPRING FOLLOW CAMERA

- Idea
  - "Store" two camera positions – ideal and actual
  - Ideal camera computed from basic follow model
  - Actual is attached on a virtual spring to the ideal, and initialized as the ideal
    - Has a position and velocity

SPRING FOLLOW CAMERA

- $x = a_{eye} - i_{eye}$
- $a = -kx - dv$
  - a is acceleration, $k \in [0,1]$ is spring constant, $d \in [0,1]$ is damper constant
- $v = v + a\Delta t$
- $c_{eye} = c_{eye} + v\Delta t$
  - Euler integration will be discussed more in Ch. 7 (physics)
  - Can apply methodology to the at/up vectors as well

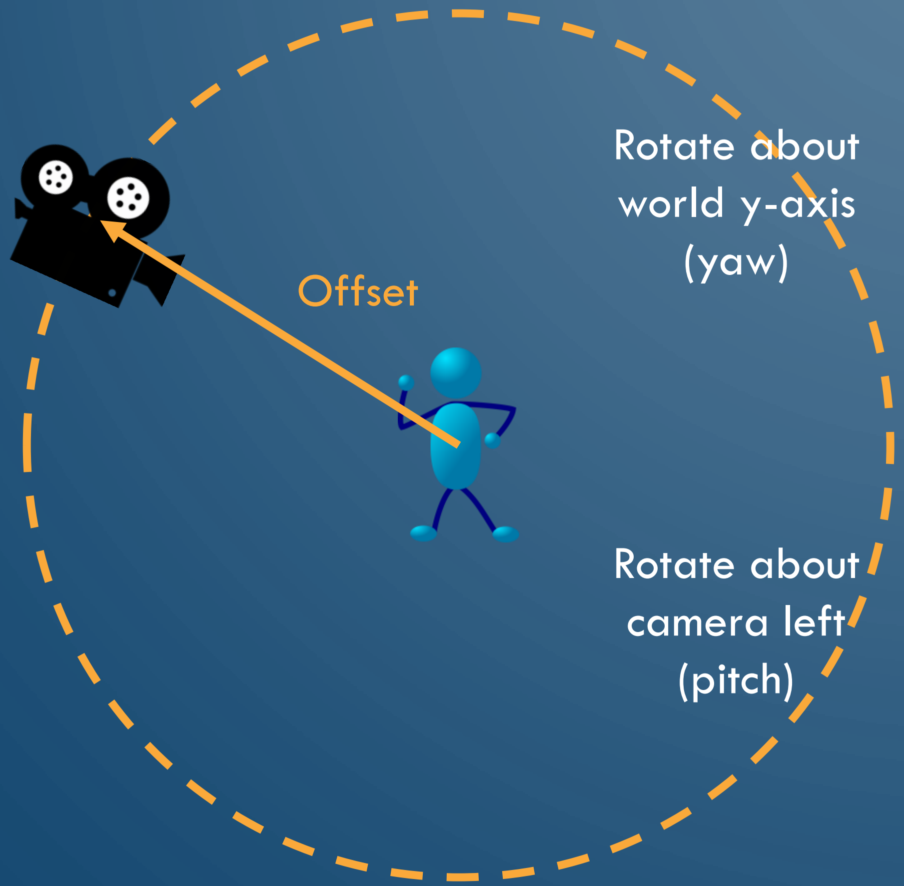
Ideal

i_{eye}, i_{at}, i_{up}

Actual

$c_{eye}, c_{at}, c_{up}, v$

ORBIT CAMERA

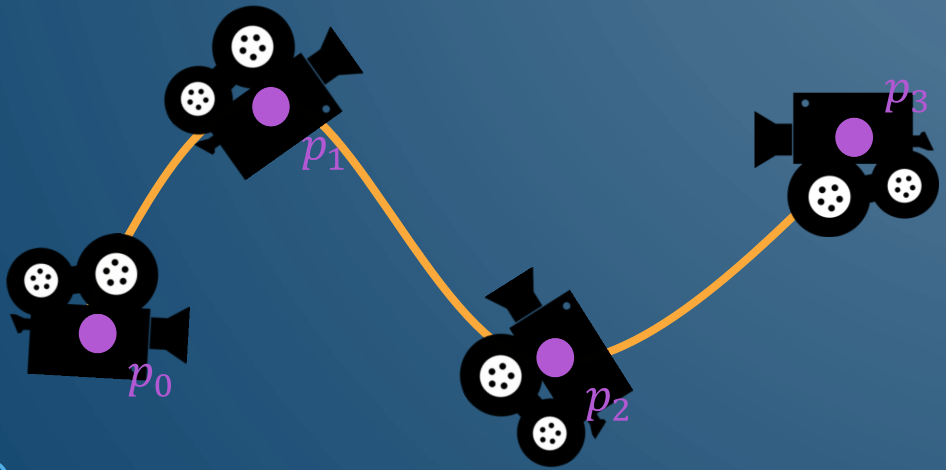


- Determine camera position change based on change in yaw and pitch
  - Can use spherical coordinates instead
- $q_{yaw} = QFromAA(w_{up}, yaw)$
- $offset = rotate(offset, q_{yaw})$
- $c_{up} = rotate(c_{up}, q_{yaw})$
- $left = normalize\left(c_{up} \times normalize(-offset)\right)$
- $q_{pitch} = QFromAA(left, pitch)$
- $offset = rotate(offset, q_{pitch})$
- $c_{up} = rotate(c_{up}, q_{pitch})$
- $c_{eye} = t_{pos} + offset$
- $c_{at} = t_{pos}$

FIRST-PERSON CAMERA

- Essentially same as orbit, except that you rotate the target position instead, so yaw and pitch are stored instead of incrementally changed
- Eye has a vertical offset from player position (ground level)

SPLINE CAMERA



- Smooth interpolation between reference frames in parametric coordinates $t \in [0,1]$
- Example spline: **Catmull-Rom** spline

$$p_t = \frac{1}{2}((2p_1 + (-p_0 + p_2)t + (2p_0 - 5p_1 + 4p_2 - p_3)t^2$$

ADDITIONAL CONSIDERATIONS

- **Camera collision**
  - Place object in front of occluding object
  - Make occluding object transparent
- **Picking**
  - Click on object in 3D world
  - Required **unprojection** of device coordinate

SUMMARY

- Discussed 2D graphics tricks and provided an overview of 3D graphics concerns
  - Remember the 3D viewing pipeline
- Overviewed some basic mathematics of camera models