• **Graphics pipeline**
  • Generates (or renders) a 2D image given a 3D scene of objects
    • I.e., whole sequence of operations that takes us from *(3D) OBJECTS ➔ PIXELS IN IMAGE*
  • Also called “*graphics rendering pipeline*” or just “*the pipeline*”
  • Composed of both hardware and software
3D SCENES

• A 3D scene contains:
  • A virtual camera – has a position and orientation (which way it’s point and which way is up), like in the image above
  • 3D Objects – stuff to render; have position, orientation, and scale; also called geometry
  • Light sources - where the light is coming from, what color the light is, what kind of light is it, etc.
  • Textures/Materials – determine how the surface of the objects should look
TRANSFORMATIONS

• We’re going to talk a lot about transformations in these slides
  • Examples of transformations:
    • Move this teapot model +5 units in X
    • I have the coordinates for this teapot relative to the WORLD origin → give me the coordinates relative to the CAMERA origin and axes
  • These will ultimately be handled using matrices
    • We’ll review matrices later as well as how to setup the transformation matrices we will need here
    • For now, though, we’ll look at what is going on at a higher level...

http://www.ranthollywood.com/wp-content/uploads/2015/05/Transformers.jpg
PIPELINE STEPS

• **Start with:**
  • 3D geometry in WORLD space

• The pipeline has four basic steps to perform:
  • **Transform** 3D points/primitives
    • MODEL or WORLD space $\rightarrow$ SCREEN coordinates/space
  • **Rasterize** primitives to screen
    • SCREEN-coordinate primitives $\rightarrow$ fragments (with interpolated attributes)
  • **Process fragments**
    • Fragment attributes $\rightarrow$ final fragment color
      • E.g., per-fragment lighting
  • **Blend fragments**
    • Fragments from multiple objects $\rightarrow$ final pixel colors
Given these steps, the pipeline can be broken up into stages:

- **Application (CPU)**
  - *Output*: either 3D geometry in MODEL/WORLD space AND/OR rendering commands

- **Vertex processing (GPU)**
  - Can be controlled/programmed by *vertex shaders*
  - *Output*: transformed geometry in SCREEN coordinates

- **Rasterization (GPU)**
  - *Output*: fragments (with attributes) for each object

- **Fragment processing (GPU)**
  - Can be controlled/programmed by *fragment shaders*
  - *Output*: fragment colors

- **Fragment blending (GPU)**
  - *Output*: final pixel colors
FIXED-FUNCTION VS. PROGRAMMABLE STAGES

• Legacy OpenGL did EVERYTHING via a fixed-function pipeline
  • No direct control → basically turn options on and off
• Newer OpenGL (3.0+) did away with almost all of the fixed-function pipeline in favor of programmable shaders
  • However, some parts of the pipeline are still fixed-function
APPLICATION STAGE
APPLICATION STAGE

• Basically your application
  • Usually where you handle collision detection, animation, physics, etc.

• Runs on CPU

• At some point, has to send geometry to vertex processing stage
  • Geometry – rendering primitives, like points, lines, triangles, polygons, etc.
  • In OpenGL:
    • OLD fixed-function pipeline $\rightarrow$ sent geometry every frame
    • NEWER pipeline $\rightarrow$ copy it over to GPU once; only update if geometry changes

• Also sends any rendering commands to GPU
  • E.g., “go render the vertices in that buffer I sent you earlier”
VERTEX PROCESSING STAGE
VERTEX PROCESSING STAGE

• Performs majority of per-polygon and per-vertex operations
• In days of yore (before graphics accelerators), this stage ran on the CPU
• Has 5 sub-stages
  • Model and View Transform
  • Vertex Shading
  • Projection
  • Clipping
  • Screen Mapping
MODEL COORDINATES

• Usually the vertices of a polygon mesh are relative to the model’s center point (origin)
  • Example: vertices of a 2D square
    • (-1,1)
    • (1,1)
    • (1,-1)
    • (-1,-1)
  • Called modeling or local coordinates

• Before this model gets to the screen, it will be transformed into several different spaces or coordinate systems
  • When we start, the vertices are (usually) in model space (that is, relative to the model itself)
• Let’s say I have a teapot in model coordinates
• I can create an instance (copy) of that model in the 3D world
• Each instance has its own model transform
  • Transforming model coordinates → to world coordinates
  • Coordinates are now in world space
  • Transform may include translation, rotation, scaling, etc.
Only things visible by the virtual camera will be rendered

- Camera has a position and orientation

- The **view transform** will transform both the camera and all objects so that:
  - Camera **starts** at world **origin** (0,0,0)
  - Camera **points** in direction of **negative z axis**
  - Camera has **up direction** of **positive y axis**
  - Camera is set up up such that the **x-axis** points to **right**

- **NOTE:** This is with the **right-hand rule** setup (**OpenGL**)
  - DirectX uses left-hand rule

- Coordinates are now in **camera space** (or **eye space**)
OUR GEOMETRIC NARRATIVE THUS FAR...

• Model coordinates $\rightarrow$ MODEL TRANSFORM $\rightarrow$ World coordinates $\rightarrow$ VIEW TRANSFORM $\rightarrow$ Camera coordinates
  • Or, put another way:
  • Model space $\rightarrow$ MODEL TRANSFORM $\rightarrow$ World space $\rightarrow$ VIEW TRANSFORM $\rightarrow$ Camera (Eye) space
VERTEX SHADING

• Our coordinates are now in camera (eye) space
• If we so desire, we can compute shading per vertex now
  • Can also output other attributes to be interpolated (normals, texture coordinates, etc.)

• Advantage of working in camera space:
  • Origin = camera origin \( \rightarrow \) don’t need to pass in camera position!
PROJECTION

• **View volume** – area inside camera’s view that contains the objects we must render
  • For perspective projections, called **view frustum**

• Ultimately, we will need to map 3D coordinates to 2D coordinates (i.e., points on the screen) → points must be **projected** from three dimensions to two dimensions

• **Projection** – transforms view volume into a unit cube
  • Converts camera/eye coordinates → **normalized device coordinates**
  • Simplifies clipping later
  • Still have Z coordinates for each point (HOWEVER, perspective projection will change them nonlinearly!)

• In OpenGL, unit cube = (-1,-1,-1) to (1,1,1)
CLIPPING

• What we draw is determined by what’s in the view volume:
  • Completely inside → draw
  • Completely outside → don’t draw
  • Partially inside → clip against view volume and only draw part inside view volume

• When clipping, have to add new vertices to primitive
  • Example: line is clipped against view volume, so a new vertex is added where the line intersects with the view volume

• We’ll discuss clipping in more detail later...
SCREEN MAPPING

• We now have our (clipped) primitives in normalized device coordinates (which are still 3D)
• Assuming we have a window with a minimum corner \((x_1, y_1)\) and maximum corner \((x_2, y_2)\)
• Screen mapping
  • \(x\) and \(y\) of normalized device coordinates \(\rightarrow x'\) and \(y'\) \textit{screen coordinates (also device coordinates)}
  • \(z\) coordinates unchanged
  • \((x',y',z) = \textit{window coordinates} = \textit{screen coordinates} + z\)
• Window coordinates passed to rasterizer stage
SCREEN MAPPING

• Where is the starting point (origin) in screen coordinates?
  • **OpenGL** → lower-left corner (Cartesian)
  • **DirectX** → sometimes the upper-left corner

• Pixel = picture element
  • Basically each discrete location on the screen

• Where is the center of a pixel?
  • Given pixel (0,0):
    • **OpenGL** → 0.5, 0.5
    • **DirectX** → 0.0, 0.0
RASTERIZATION STAGE
RASTERIZATION STAGE

• **Rasterization stage**
  
  • *Enumerates* pixels covered by primitive
  
  • *Interpolates* values (called **attributes**) across the primitive
    
    • Color would be an attribute
    
    • Also interpolates Z values!
  
  • **Outputs fragments**
    
    • One fragment for each pixel covered by primitive
    
    • Each fragment “lives” at a particular pixel
    
    • Each fragment has its own set of attribute values
FRAGMENT PROCESSING STAGE
FRAGMENT PROCESSING STAGE

• **Fragment processing stage**
  • Compute color for each fragment $\rightarrow$ per-fragment shading
  • Can be controlled/programmed by **fragment shaders**

• **Example on right:**
  • Given the following attributes:
    • Interpolated diffuse color
    • Interpolated normals
    • Light position
  • Output per-fragment color
FRAGMENT BLENDING STAGE
FRAGMENT BLENDING STAGE

- **Color buffer** → stores color for each pixel

- **Fragment Blending stage**
  - Combine each fragment color with color current stored in color buffer
    - May replace it altogether OR blend the colors (alpha blending)
WHO’S ON FIRST?

- Given two objects, how do we make sure that the object in front gets drawn over the object in the back?

- Two options:
  - Painter’s algorithm
  - Z-Buffer algorithm
PAINTER’S ALGORITHM

- **Painter’s algorithm** = draw triangles in back-to-front order
  - **Problems:**
    - What happens if objects intersect each other?
    - What if you have an **occlusion cycle**?
    - Sorting all triangles in scene \(\rightarrow\) SLOW

- Intersection
- Occlusion Cycle
Z-BUFFER

• Z-buffer
  • Pretty much the standard for depth sorting in almost all applications
  • Basic algorithm:
    • Check z value of incoming fragment
    • If closer to camera than previous value in Z-buffer → override color in color buffer and update z value
Z-BUFFER ALGORITHM

- Let’s assume we are dealing with OPAQUE surfaces
  - I.e., no transparency
- Have **depth buffer** = separate buffer that holds depth values
  - Initialize to 1.0
    - *Remember:* in normalized device coordinates, max value = 1.0
  - If projected fragment $z < \text{depth}[x,y]$:
    - Depth$[x,y] = z$
    - Color$[x,y] = $fragment’s color
  - Otherwise, ignore fragment

Note: Both incoming fragments are trying to write to the upper-left corner of the color buffer.
Z-BUFFER: PROS AND CONS

• **Advantages:**
  - $O(n) \rightarrow n = \text{number of primitives}$
  - Simple $\rightarrow$ easy to implement in graphics hardware
  - Can draw OPAQUE objects in ANY order!

• **Disadvantages:**
  - Does not work properly with transparent or semi-transparent surfaces (if drawn out of order)
  - Can be a little inefficient $\rightarrow$ might already have nearest pixel, but still have to check every other overlapping polygon on that pixel

• Also, there are issues with the **precision** of z-buffer must be considered
  • We’ll revisit this after we look at the perspective project matrix later...
WRITING TO THE BUFFER

• **Frame buffer**
  • Means all buffers on system (but sometimes just refers to color + Z-buffer)

• To prevent the user from seeing the buffer while it’s being updated \(\rightarrow\) use **double-buffering**
  • Two buffers, one visible and one invisible
  • Draw on invisible buffer \(\rightarrow\) swap buffers
REFERENCES

• Many of the images in these slides come from the book “Real-Time Rendering” by Akenine-Moller, Haines, and Hoffman (3rd Edition) as well as the online supplemental material found on their website: http://www.realtimerendering.com/

• Some also are from the book “Computer Graphics with OpenGL” by Hearn, Baker, and Carithers (4th edition)