Game Graphics & Real-time Rendering

CMPM 163, S2019

Prof. Angus Forbes (instructor)
angus@ucsc.edu

Manu Thomas (TA)
mthomas6@ucsc.edu

David Abramov (TA)
dabramov@ucsc.edu

Website: creativecoding.soe.ucsc.edu/courses/cmpm163_s19
Slack: https://ucsccmpm163.slack.com

Class information

Class website:

https://creativecoding.soe.ucsc.edu/courses/cmpm163_s19

Slack is the main form of class communication:

https://ucsccmpm163.slack.com

Our TAs are Manu Thomas and David Abramov

They will lead the lab sections starting next week

For Thursday

Download and explore Unity:

- Make sure you download and test Unity on your laptop or a lab computer: https://unity3d.com/get-unity/download (the stable version is “2018.3”, and we are using the “personal” version). Unity should work on any recent laptop or desktop, but runs better with a fast GPU.

- Follow the introductory tutorials at: https://unity3d.com/learn/beginner-tutorials

- Get familiar with the Unity Manual: https://docs.unity3d.com/Manual/index.html

Setting up Unity

If you haven't used it before, takes a little bit to get used to, but it simplifies a lot of the boilerplate related to setting up a new scene, etc.

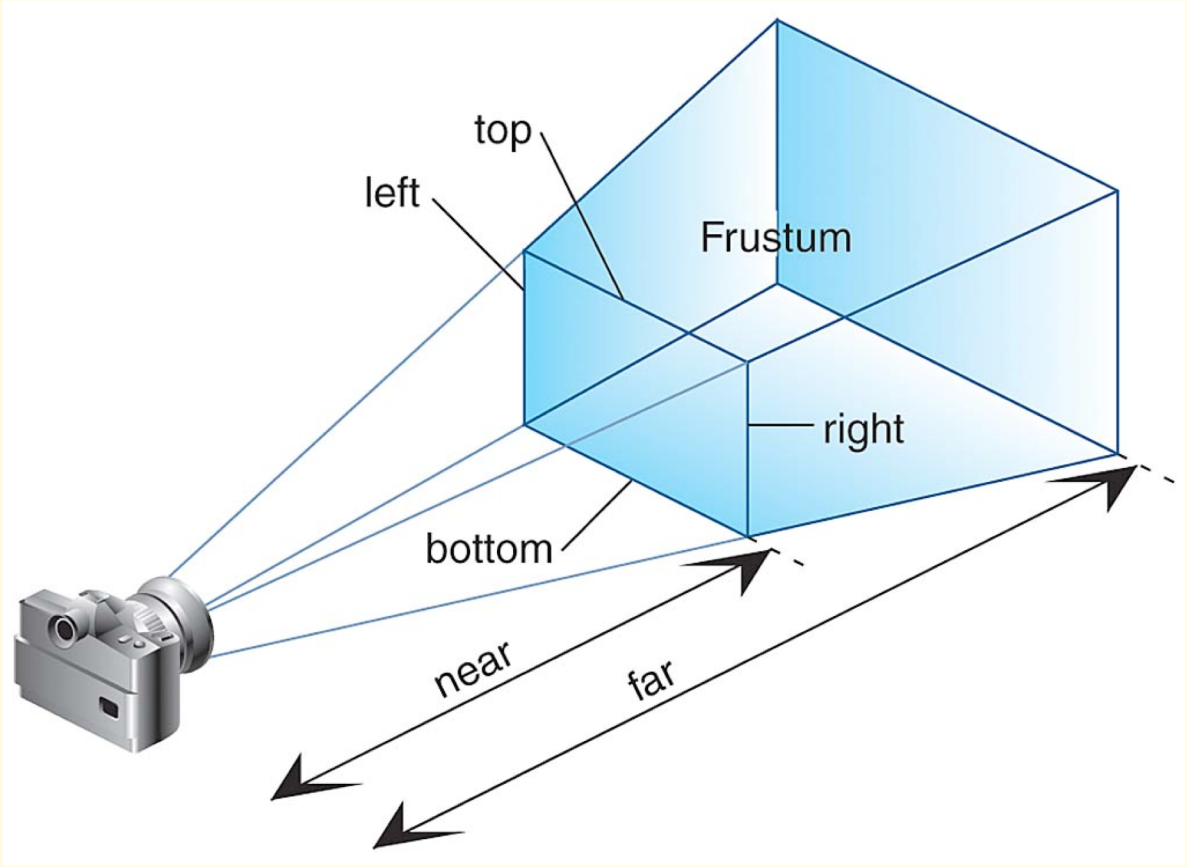
It includes a ton of functionality, but we will start by looking at parts that let us control how particular objects are "shaded", or rendered to the screen.

3D Scene

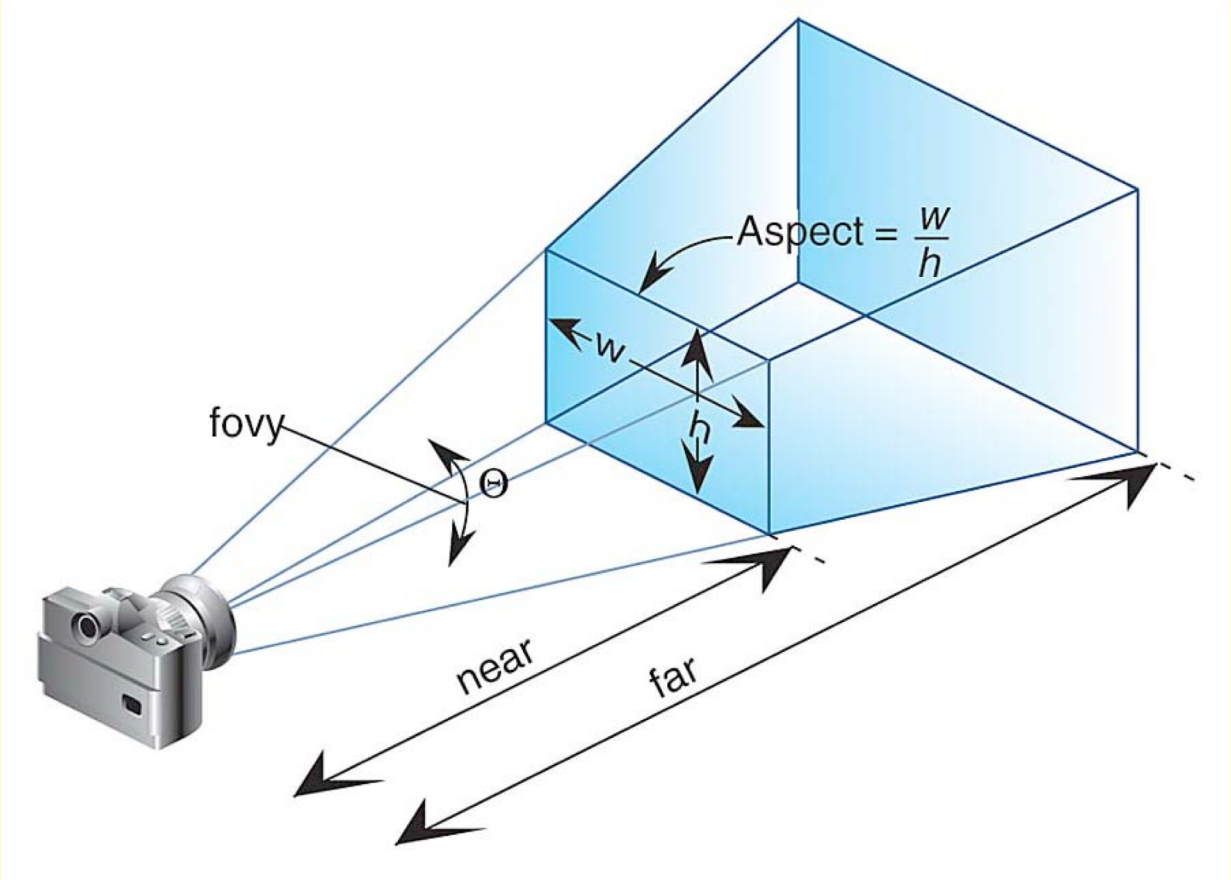
In most graphics frameworks, a scene consists of:

- **Cameras:** A camera characterizes the extent of the 3D space that will be projected onto the 2D image plane
- **Lights:** Different positions, orientations, colors, and types of light are used to illuminate the scene
- **Geometry:** Objects made out of triangles populate the scene, and are lit by the different light source
- **Materials:** The objects in the scene can be characterized by different material properties that respond to light
- **Textures:** The objects can also have different textures, or images, placed on them

Perspective camera



Perspective camera



Rendering pipeline

The rendering pipeline describes the series of operations that transform your programmatically defined 3D scene into an actual 2D image.

This is done using shaders.

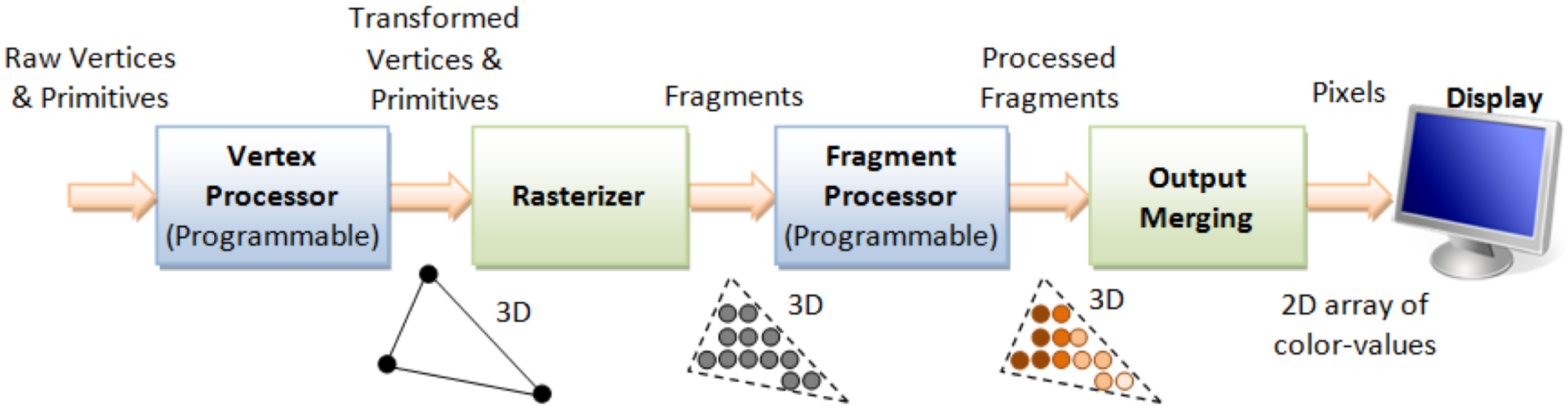
- **Vertex shader:** The vertex shader is responsible for turning 3D geometry into “normalized device coordinates” (2D values between -1 and +1, plus a depth value)
- **Fragment/pixel shader:** The fragment shader is responsible for coloring in each of those pixels, and then outputting it on the 2D screen

In Unity, each shader program lives in its own function. At the start of an application, these files are compiled into a "program" and copied over to the GPU, along with texture data.

When the application is running, usually at 60fps, during *each frame*, the following step occur:

- 1. The shader program is activated on the GPU ("bound")
- 2. Any texture data you will use is also bound (ie, made available to the shader)
- 3. 3D points describing your geometry are passed in to the GPU, and input into your vertex shader, one triangle at a time
- 4. The vertex shader projects the triangle into a simpler coordinate system and outputs it as "fragment," or pixel data, which is input into the fragment shader
- 5. The fragment shader decides what color to make each pixel, and then draws it on the screen

Perspective camera



3D Graphics Rendering Pipeline: Output of one stage is fed as input of the next stage. A vertex has attributes such as (x, y, z) position, color (RGB or RGBA), vertex-normal (n_x, n_y, n_z), and texture. A primitive is made up of one or more vertices. The rasterizer raster-scans each primitive to produce a set of grid-aligned fragments, by interpolating the vertices.

Rendering pipeline

Let's look at a high level description of this process, using **Unity**. (Most graphics/game frameworks conceptualize this process similarly.)

This is done using **shaders**.

- **Vertex shader:** The vertex shader is responsible for turning 3D geometry into 2D pixels
- **Fragment/pixel shader:** The fragment shader is responsible for coloring in each of those pixels

Unity shaders

The most basic vertex shader is called a “pass-through shader”

This vertex shader does only this:

It takes each triangle used to define your geometry and projects it into 2D, then passes this 2D data to the fragment shader.

The most basic fragment shader simply gives each pixel a color value.

uniform: data that is static over the life of the binding – It will be the same for all geometry passed in the shader. Uniform variable are available to both the vertex and the fragment shader.

attribute: data that is defined per vertex for each geometry that is passed in with each mesh (i.e., position, normal, texture coordinates). Attribute data is only available in the vertex shader.

varying: this is how you link data from the vertex shader to the fragment shader.

A shader program in Unity looks a lot like a C program, with some differences in syntax.

In its “vert()” function, the vertex shader **must** return a variable (or a struct that contains this variable) of type **SV_POSITION**, which is used by the fragment shader

In its “frag()” function, the fragment shader **must** return a variable of type **fixed4**, which defines the RGBA values for a pixel.

Let’s look at our shader programs… (see demo code)

Get code to run on your laptop

Customize the code (without breaking it!)

- Change the color values and see what happens for the Color shader
- Load in a new image instead of the default one for the Texture Shader

Questions?

- Homework package #1 will be sent out on this weekend
- Lab sessions will start next week (led by Manu on Monday and David on Wednesday and Thursday)