Graphics Foundation

* Most video games are created using polygons (mainly triangles) for all textures, characters, structures, etc.
* Early games would use bitmaps to color in the pixels
  + Calculate where the pixels would be on the screen
  + Got more complicated as resolution got bigger, lighting got more advanced, and shading got more complex
* Software rendering
  + Ram->Video Card->Comp Screen
  + 2560\*1600 = 4,096,000 pixels => ~12mb
  + Sizeof(War and Peace)
  + 60 times a second
* Arcade games
  + Rendered Data onto Hardware skipping the video card step
  + 3D games before 3D on computer (more colorful and brighter)
* 1981/1982
  + First home console, very colorful graphics that ran quickly
  + Storing data on hardware
* Map and Distort Images to fit onto a 3D landscape.
  + Most efficient way of mapping images to polygons is triangles
* Late 1980s to 1990s
  + Gams were being programmed to be ran on the GPU allowing for more graphically intensive and complex
  + Just tinier triangles stored on the video card
* Hardware Rendering
  + Only copy the data needed
  + Ram->Video Card->Monitor
* No difference between 2D and 3D rendering

GPU Pipeline

* Two triangles, make up a square, in which the sprites are mapped
* Vertex Data->Vertex Shader->Fragment Sader( also receiving textures)->Rasturize
* Vertex Data
  + Defining the points that make up the triangle
  + A **polygon** is defined by points in space called **vertices**
  + Every point has a position on the cartesian plane
  + For every triangle, 6 numbers that describe the position of the vertices. (Focusing on 2D)
  + Order matter in how the triangle is described
    - Optimization = one side is invisible and doesn’t render the textures
  + Define them in counter-clockwise order
  + Screen = Area GPU is rendering
    - Measured in Normalized Device Coordinates
    - Aspect Ratios
* Vertex Shader
  + A program that transforms the attributes (such as position, color or others) of every vertex passed to the GPU
* Fragment Shader
  + A **program** that returns **the color of each pixel** when geometry rasterizes onto the **screen**.
* Textures
  + Should be processed along with the fragment due to fragment is working with colors
* Vectors
  + Direction 1-dimension: +1 or -1 on the x-axis
  + Direction 2-dimensions: 1,0,-1 x-axis; 1,0,-1 y axis in a matrix []
    - Changing numbers can either scale or rotate
* Homogenous Coordinates
  + Take a unit system and take it up 1 dimension (XY1)
  + Used for skewing, transformations and projection mapping objects
* Transformation matrices: Class Slide for images & examples
* Modelview Matrix ->Vertex Shader
* Projection Matrix-> Vertex Shader
  + A means representing a **three-dimensional** object in **two-dimensions**
* Perspective vs Orthographic Projection
  + Perspective much like normal eye sight
  + Othographic, squishing everything down to a 1:1 box
    - Defines what you see in your world

OpenGL (In-Class OpenGL-ES)

* Open a new window, setting the dimensions
* void glViewport (GLint x, GLint y, GLsizei width, GLsizei height);
  + Sets the size and offset of rendering area (in pixels)
  + Splitscreen: render the same game twice on the top and bottom of screen
* Shader Program class “ShaderProgram.h”
  + ShaderProgram program”RESOURCE\_FOLDER”vertex.glsl”, RESOURCE\_FOLDER”fragment.glsl”);
  + Combination of the vertex shader and fragment shader
  + Made specifically for this class
* Prefix RESOURCE\_FOLDER makes it easier to collaborate due to simple change to value of variable
* Matrix Class “Matrix.h”
  + Matrix projectionMatrix; Matrix modelviewMatrix;
  + Pass the matrices to our program
* void Matrix:SetOrthoProjection (float left, float right, float bottom, float top, float near, float far);
  + Sets an orthographic projection in a matrix
  + Setting box (near and far) only -1 to 1
* Drawing Polygons (HAPPENS EVERY FRAME)
* void glUseProgram (Glint programID);
  + Use the specified program id. We need to tell OpenGL what shader program to use before we render anything
* void glVertexAttribPointer (GLint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const GLvoid \*pointer);
  + Defines an array of vertex data (counter clockwise)
  + Vertices = 6 floats in an array (pointer is pointing to array of vertex data)
* glDrawArrays (GLenum mode, GLint first, GLsizei count);
  + Render previously defined arrays
* void glDisableVertexAttribArray (GLuint index);
  + Disables a vertex attribute array
* SDL\_GL\_SwapWindow(displayWindow)
  + Swap between the screen that’s rendering and the screen shown
* void Matrix::Translate (float x, float y, float z);
  + Translates a matrix by specified coordinates
* void Matrix::Scale (float x, float y, float z);
  + Scales a matrix by specified coordinates
* void Matrix::Rotate(float radians)
  + Rotates a matrix by specified radians
  + Convert degrees to radians (RADIANS = DEGREES \* (PI/180)
* Matrix multiplication is not commutative, so the order of matrix operations matters
  + Personal order: scale, translate, rotate