Hello Triangle 프로그램으로 기본코딩을 시작해보자~

# 1. Hello Triangle

html의 onload에 해당하는 함수는 script src로 html에 포함시킨 js파일에 존재하는 함수이다.

testGLError -> 디버깅에 사용하는 GL Error를 테스트하는 함수

intialiseGL(canvas)

* getContext를 통해 webgl을 사용하겠다는 걸 알림
* initialiseGL 하기 전에 GL function call 되면 오류 뜸
* || 를 통해서 webgl을 지원안하는 버전이면 || 이후의 module을 사용함
* gl.viewport 함수로 draw area를 설정한다. // 그릴 부분을 설정하는 것 // 만약에 200, 200으로 설정하면 좌측하단 0, 0을 기준으로 아주 작은 영역에 비율을 맞춰 그림이 그려진다.

initialiseBuffer()

* canvas의 상대좌표를 이용하여 vertexData를 선언하는 곳 // 중앙이 origin이다.
* createBuffer 이후에 bindBuffer함수로 어떤 Buffer를 사용할건지 GPU에게 알린다. Buffer에는 두 가지가 있는데, ArrayBuffer & ElementBuffer이다.
* bufferData로 보내줄 때 Float32Array / STATIC방식으로 GPU에게 보내면 GPU는 이를 효율적으로 보관한다.

initialiseShaders()

* fragmentShader / vertexShader가 순서에 맞춰 작동됨
* Pipeline에 따라 vertexShader에는 uniform, attribute등이 input으로 들어가고 gl.position이 갱신된다. fragmentShader에서는 fragcolor가 set된다.
* vertexShader까지는 vector // fragmentShader에서는 bitmap이다.

renderScene()

* 1차원 배열로 이루어졌지만 4\*4 변환행렬을 표현하는 Matrix가 있다.
* gl.clearColor 는 background color이다.
* transformationMatrix는 CPU에 저장되어 있다가 vertexShader의 uniforms으로 전송된다.
* drawArrays 함수로 gl.TRIANGLES를 사용하면 123 / 456 / 789… 와 같은 vertex끼리 하나의 삼각형을 만들게 된다

main()

* html에 의해 실행된다.
* GL -> Buffer -> Shaders 순서로 실행
* requsetAnimFrame 객체는 함수 객체이다
* window.requestAnimationFrame을 반환하는데 얘는 브라우저에게 canvas에 새로운 걸 그렸으니 업데이트를 시키라고 하는 역할
* renderLoop에서 재귀호출을 통해서 종료까지 계속 그림

# 2. VBO Overview

Vertex Buffer Objec를 어떻게 메모리에서 그래픽메모리로 보내는지

Vertex Data 는 CPU memory에 저장되어 있다가 GPU memory로 이동한다.

어레이버퍼 – vertex attributes에 쓰임

엘리먼트 어레이버퍼 – 어셈블리에 쓰이는 indexes of primitive // 2 3 4 vertex로 triangle만드는 방식에 쓰임

OpenGL ES 와 Web GL 을 비교해서 보여줌 // Web GL이 사용하기 더 편하다.

initialiseBuffer 함수에서 VBO를 create하고 binding한다.

createBuffer / bindBuffer / bufferData -> testError

* 함수 3번 호출 후에 에러테스트는 1번만 한다.
* 함수가 비동기적으로 호출되고 큐에 들어간 후에 testError가 실행되므로 가장 최근의 에러만 확인한다.
* 어떤 부분에서 에러가 났는지 정확히 알 수 없으므로 로그를 분리하여 확인한다.

gl.drawArrays의 parameter를 통해 vertexData 읽는 위치를 설정한다.

int first, long count 형식이다 // 0 6 이면 0번 인덱스부터 6개 읽는다는 의미

ELEMENT\_ARRAY\_BUFFER이용해서 삼각형 그리는 방법 소개

# 3. Vertex Attribute

vertex buffer object 사용하는 법을 배운다

Attribute는 Vertex Shader의 input으로 사용된다. // 각 그룹마다 다른 input을 가진다.

Uniform은 common value로서 모든 vertex가 같은 값을 공유한다.

gl.MAX\_VERTEX\_ATTRIBS 를 8 이상으로 유지하면 스마트폰 같은 환경에서 실행되지 않을 수 있으므로 8이 적당, 8이하라면 Web GL을 지원하는 HW가 아니다.

* 8이 스탠다드다

void bindAttribLocation 함수를 이용하여 Attribute location을 shader에 set한다.

void vertexAttribPointer(uint index, int size, enum type, bool normalized, long stride, long offset)

normalize : -128 ~ 127 을 -1 ~ 1 의 값으로 표준화 시킴

stride : 다음 vertex까지의 거리

offset : 상대주소값(?)

x, y, z은 하나의 attribute이다.

Attributes : Position \* 3 / Color \* 4 / ST \* 2 / UV \* 2 가 있을 때

Array of structure -> 전부 나열

Structure of Array

* A = [x1, y1, z1, x2, y2, z2 , …]
* B = [r1, g1, b1, a1, r2, g2, b2, a2, …]

Array of structure 방법이 더 낫다. -> 퍼포먼스와 관리가 쉽다.

Structure of Array 방식을 사용하면

AttributeLocation을 따로 설정해줘야함 // source에서도 따로 assign해야 함

bindBuffer 와 enableVertexAttribArray도 따로 설정해줘야함

* 여러 개의 attribute을 사용하는 것은 복잡..

# 4. Primitive Assembly

Primitive aseembly는 Rasterization의 전 단계

3종류의 primitive가 있다.

1. point / 2. line / 3. triangle

drawArrays / drawElements의 primitive types

* enum type / POINTS / LINES, LINE\_STRIP, LINE\_LOOP / TRIANGLES, TRIANGLE\_STRIP, TRIANGLE\_FAN

GL\_POINTS

LINE

* LINES : n vertices – n/2 lines
* LINE\_STRIP : n vertices – (n-1) lines, (v0, v1) (v1, v2) (v2, v3) …
* LINE\_LOOP : n vertices – (n) lines, 점을 전부 이어서 연결

TRIANGLE

* GL\_TRIANGLES : n vertices – (n/3) triangles
* GL\_TRIANLGES\_STRIP : n vertices – (n-2) triangles (v0, v1, v2), (v1, v2, v3) …
* GL\_TRIANGLES\_FAN : n vertices – (n-2) triangles (v0, v1, v2), (v0, v2, v3), (v0, v3, v4) …

primitive assembly는 Clipping -> Perspective Division -> Viewport Transformation 프로세스로 이루어진다.

Clipping은 그려야 할 것이 screen 범위 내에 있는지 판단하는 역할

Perspective Division : 3D 그래픽에 사용됨 // x,y,z,w 에서 w를 이용하여 보이는 크기 변경

(x/w, y/w, z/w) 로 각 좌표를 w로 나눈다. w가 0일 때는 1로 판단하여 계산한다 -> x,y,z가 그대로 그려짐

Viewport Transformation : drawing area를 설정하는 함수 // 이 함수에 의해 viewport coordinate, normalize coordinate 로 변경된다. 이후에 Rasterization으로 보내짐

primitive assembly는 컨트롤할 수가 없고 자동으로 결과가 도출된다.

drawArrays로 큐브를 그리면 12개의 triangle -> 36 vertex 가 필요하지만

drawElement로 큐브를 그리면 8개의 vertex만 필요하다.

# 5. Shader Overview

openGL EL 1.0은 shader language를 쓰지 않았다. 2.0 부터 Shader language를 사용했다.

GLSL 1.5ver 부터는 Attirb / varying 이 아니라 in / out으로 사용함

Shading language는 Parallel Programming (SIMD)로 실행됨

같은 instruction을 다른 data에 실행

2개의 shader는 각각 하나의 프로그램이다. 이들을 실행하기 위해 shader language가 필요

Shading Language를 사용하려면

1. vertex / fragment source가 필요

2. 각 shader를 compile

3. program – create / attach / link / use

data type / reserved keyword는 c와 비슷하다

matrices가 다르니 조심!

vec2는 2개의 component가 있는 vector // vec3 .. vec4 ..

mat2는 2\*2 matrix // mat3 .. mat4 ..

matrix-vector multiplication

matrix는 column major이다. 우리가 사용하는 행렬곱셈과 같지만, 1차원배열로 바꾼다면 순서가 달라진다. [0][0] 에서 [1][0]으로 column을 기준으로 순서가 정해짐

* column / row major는 메모리 상에 어떻게 저장되느냐의 문제이지 행렬곱셈하는 방식은 같다.

m \* v 에서는 vector가 column으로서 곱해짐 -> WebGL에서 우리가 사용할 방법

post multiplication을 사용함

v \* m 에서는 vector가 row로서 곱해짐

Pointers

GLSL에는 포인터가 없다.

matrices / vector들이 기본 타입이기에 GLSL 함수들의 output으로 전달될 수 있어서!

Qualifiers

Storage Qualifiers - const, attribute, uniform, varying

별로 안중요

Vertex attributes / varying values는 fragment shader에서 interpolated 된다.

OpenGL ES 2.0는 반드시 최소한 8개의 attribute를 지원함

mat4에서는 4 \* vec4랑 같으므로 4개의 attribute를 쓸 수도 있긴 하다

* 뭔 개소리야? 그래서 된다는거야 만다는거야 ㅋㅋ

vertex shader / vertex processing unit에서는 모두 같은 uniform값 사용

* uniform은 shader에서 값이 바뀔 수 없다.

vertex processing unit은 각각 다른 attribute 사용

< Varying Qualifier >

Automatically interpolated by the rasterizer

vertex shader -> fragment shader로 전해지는 variable이다.

< Parameter Qualifiers >

In(default) / Out / Inout

< Precision Qualifiers >

highp – for vertex shader // 높은 quality

lowp – for color value // 0~1.0까지의 값을 가짐, 메모리 낭비 줄이기 위한 Qualifier

medium – somewhere in // 적당한 정확도를 가짐, 정확히 어느정도인지는 모르지만

lowp <= medium <= highp 임

lowp 8 bit / mediump 16 bit / highp 32 bit

* floating point 뿐만 아니라 int에도 쓸 수 있는데 lowp는 int로 못 씀
* int는 보통 highp의 precision이 default로 사용됨

생성자 // 37:35 부근

mat3(2.0) 하면

2 0 0

0 2 0

0 0 2

가 생성됨

built-in variables

gl\_Position / gl\_PointSize / gl\_FragColor / gl\_FragData

built-in constant

gl\_MaxVertexAttribs / gl\_MaxVaryingVectors / gl\_MaxCombinedTextureImageUnits

gl\_MaxFragmentUniformvVectors / gl\_MaxVertexUniformVectors / gl\_MaxVertexTextureImageUnits / gl\_MaxTextureImageUnits / gl\_MaxDrawBuffers

# WebGL 06. Transformation

transformation matrix의 x y 값을 바꿔 확대 //

toggle animation키를 추가해서 flag를 이용하여 on/off하는거 보여줌

< Transformation >

1. Linear

translation : orientation의 이동

Euclidean : 크기 그대로 이동

similarity : 닮음

affine : 두 개의 평행한 직선이 이동해서도 평행을 유지하면 affine이라고 함

projective : 두 개의 평행선을 이동했는데 평행성질이 바뀌는 경우 projective

affine은 projective의 특수한 경우라고 볼 수 있다.

(perspective view를 표현할 때도 이 방법이 사용됨, perspective 방법으로 불리기도 함

🡪 Linear Transformation은 오로지 vertex의 변화로 조정함 // vertex shader만 관여

🡪 vertex shader는 Linear Transformation만 관리

🡪 Linear Algebra equation으로 표현될 수 있음

🡪 대부분의 transformation은 matrix \* vector로 표현됨

2. Non-Linear

🡪 수축시킨거 같은 transformation : PINCUSHION DISTORTION

🡪 팽창시킨 transformation : BARREL DISTORTION

GPU성능이 좋다면 barrel distortion의 결과가 좋음, barrel distortion은 fragment shader의 성능을 drop시킴

Transformation은 보통 Vertex Shader에서 일어남 / Fragment Shader에서도 일어날 수 있지만 Fragment shader에서는 vertex의 position을 변경하는 곳이 아니므로 복잡하다. 여기선 안다룸

Vertex Shader에서 Transformation -> Rasterization 2가지 과정을 거치는데 Transformation 과정에서 입력된 2개의 vertex값이 transform되어 position이 바뀜

Translation matrix

1 0 0 d(x) x

0 1 0 d(y) y

0 0 1 d(z) 과 원래의 position을 나타내는 벡터 z 와 곱해짐

0 0 0 1 1

translation은 homogeneous coordinates를 이용하여 x y z w 와의 곱으로도 나타낼 수 있다.

Scaling matrix

각 축에 대해 축소 or 확장

s(x) 0 0 0

0 s(y) 0 0

0 0 s(z) 0

0 0 0 1

Rotation (2D)

x’ = x cos@ - y sin@

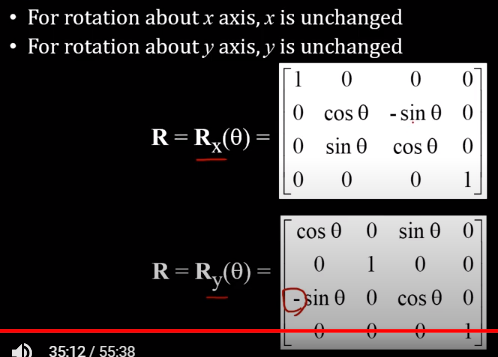
y` = x sin@ + y cos@

Rotation about z axis (3D)

x’ = x cos@ - y sin@

y` = x sin@ + y cos@

z’ = z

about x axis 는

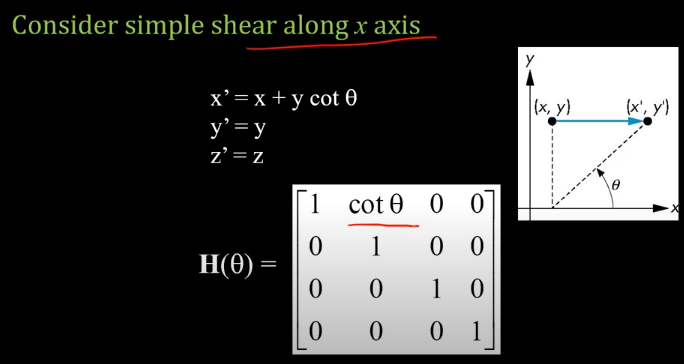
Reflection

scaling matrix에서 S(x), S(y), S(z)의 조합에 따라 바뀐다

y축 반전 / 원점 반전 / x축 반전 등이 된다.

Shear

2D로 보면 직사각형을 평행사변형처럼 눌리게 하는 transformation임

about x-axis 이면 x의 값이 변화한다

affine 변환 : 평행성 유지!

Inverses transformation

변환행렬의 역행렬

행렬곱에는 결합법칙이 성립, 행렬곱을 homogeneous coordinate로 하면 효과적

# Web 07 – coding of Transformation

gl-matrix의 함수 소개 및 실제 코딩

시험문제 example

구글시트로 개인정보 제출하고 시작

pipeline에 대한 문제 / 각 module, component에 대한 역할 설명할 수 있어야 함

input-output data에 대해서 / module별 function /

column major

Post Script transformation

그림보고 code 작성해야 함 // post script로만

perfragment diagram

# Web 08. Viewing and Projection Transformation

perspective projective은 affine변환이 아니다

카메라를 옮기는 거 -> viewing transform

model view proejection -> MVP 방식 // 곱하는 방식 조심!!

PVM 순서로 적용해줘야 한다.

컴퓨터에서 사용하는 카메라는 바늘구멍카메라(Pin hole Camera)

rotation은 순서에 따라 달라짐 // 팬 틸트 롤은 순서 바껴도 됨

scaling은 순서 상관없음

V(view) \* M(model) \* apposition = 피사체의 위치

노말벡터(up vector)는 카메라 위치 달라져도 보통 0 1 0 으로 정하고 해도 됨

transform matrix의 마지막 열이 0 0 0 1 이면 affine transform이다.(affine transform은 perspective transform에 속한다)

orthographic projection (평행투시법)

* 실물 크기 그대로 필름에 투영

원근법

near / far 필름?

# Web 09.(a) per Fragment Operation

per Fragment Operation은

Pixel Ownership test / Scissor test / Stencil test / Depth test / Blending / Ditherring

모든 GPU는 z-buffer algorithm을 통해 hidden-surface method를 사용한다.

z buffer는 한 픽셀당 최소 16bit를 필요로 함

far plane = 1.0 / near plane = 0.0

z buffer의 첫 값은 clearDepth함수로 바꿀 수 있음 보통은 1.0 //

plane을 만나면 z buffer값과 비교를 해서 더 작으면 해당 fragment는 update 아니면 그대로

z buffer의 값과 비교하여 어떤 상태일 때 업데이트 할건지

depthfunc에서 조절 가능

* NEVER / ALWAYS / LESS / EQUAL / LEQUAL / GREATER / GEQUAL / NOTEQUAL

drawing을 하기 전에 buffer를 비우자

# Web 09(C) per-Fragment Operation

r g b a 는 각각 8bit