6/6/2011

3D Basic & OpenGLES 2.0

thuy.vuthiminh@gameloft.com

phong.caothai@gameloft.com

khiem.tranthien@gameloft.com

tam.la@gameloft.com

Content

- Introduction
- Rendering pipeline
- Shader
- Basic GLSL-ES
- Basic Math
- MVP matrices
- Textures
- Obj model
- Shader effect: Skydome using cube mapping

Basic knowledge in Opengles 2.0?

Content

- **Introduction**
- Rendering pipeline
- Shader
- Basic GLSL-ES
- Basic Math
- MVP matrices
- Textures
- Obj model
- Shader effect: Skydome using cube mapping

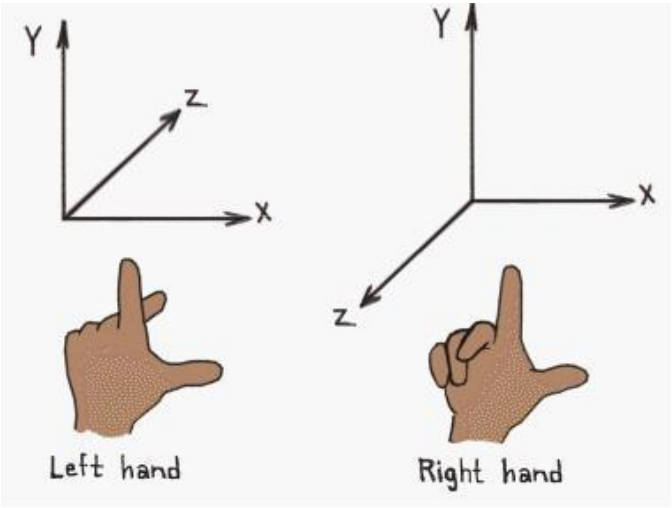
Introduction

- 3D Coordinates
- Vertex
- Edge
- Triangle & Quad
- Normal vector
- Pixel
- Texel
- Fragment
- Parallel & perspective projections
- Bonus: Color channel

3D Basic concept?

Introduction: 3D Coordinates

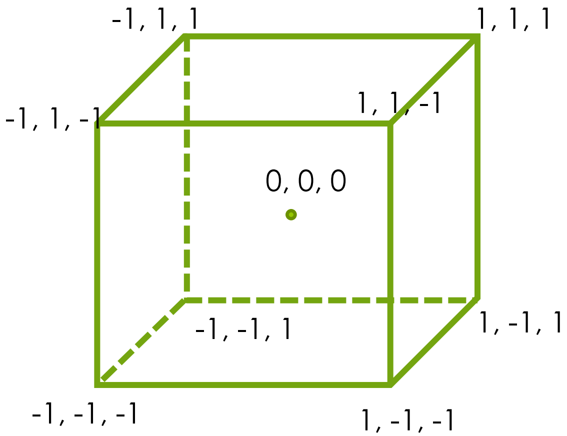
- The right handed coordinate system is used by OpenGL
- The left handed coordinate system is used by DirectX



Introduction: 3D Coordinates

The OpenGL Coordinate System

- 3D point in clip coordinates is mapped to a cube (NDC) (**N**ormal **D**evice **C**oordination)
- NDC uses ***left-handed*** coordinate system
- In the bounds of [-1,-1,-1] and [1, 1, 1]
- Everything outside that bound will not be taken into consideration.

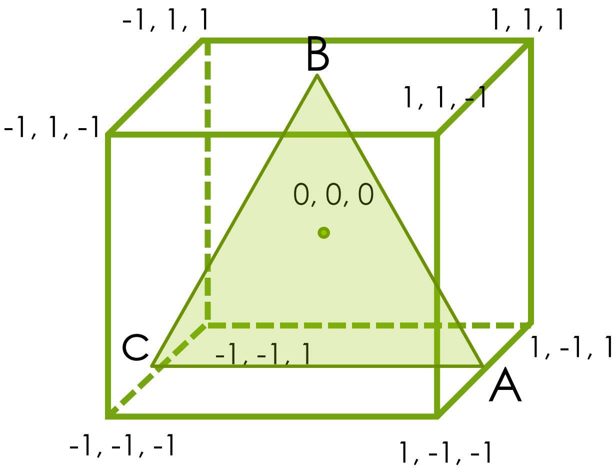


Ex: The triangle is defined by the points

A (1.0, -1.0, 0.0),

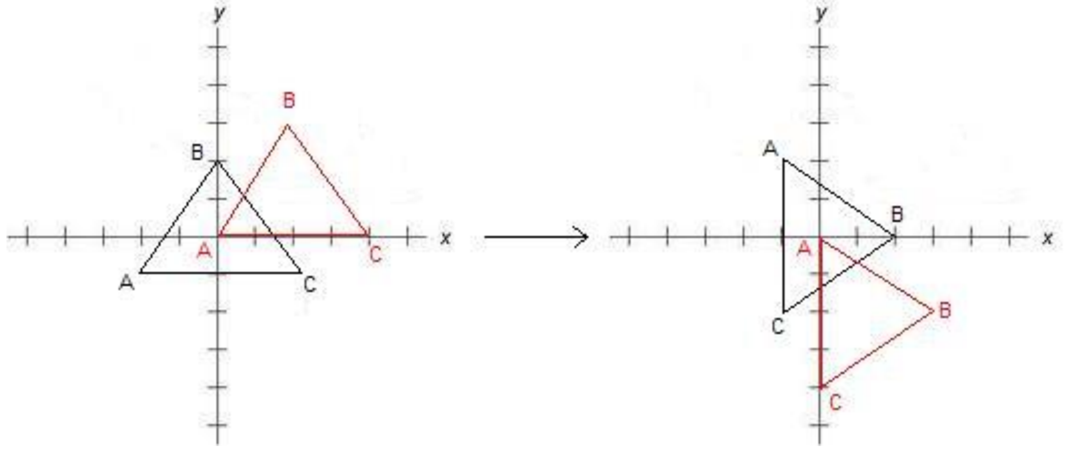
B (0.0, 1.0, 0.0),

C (-1.0, -1.0, 0.0)



Introduction: 3D Coordinates

- Every geometry, can be a triangle or a object composed from multiple triangles has a pivot.
- The pivot is the point around which the triangle will rotate
- Object space the pivot is always in (0,0,0)



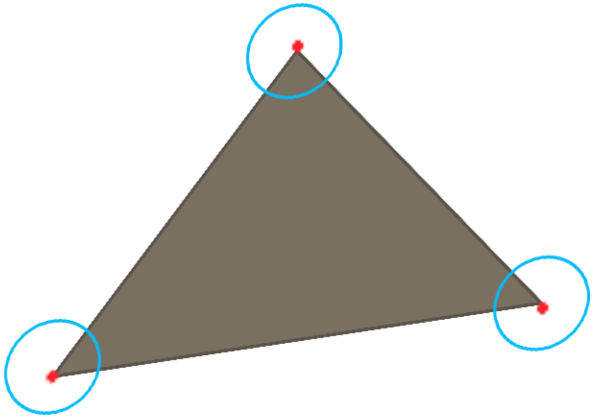
90^0 degrees rotation around the pivot.

- The black triangle has his pivot in the centroid
- The red triangle has his pivot in the point A.

Introduction: Vertex, Edge

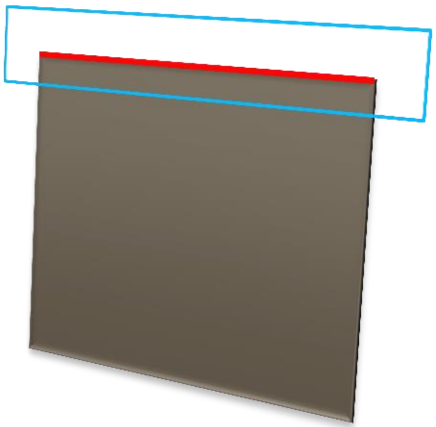
Vertex

- A vertex is a point in 3D space
- Contains:
  - Position: x, y, z
  - Normal *(later)*
  - Color
  - Texture coordinates
- Transferring data from RAM → GPU is expensive. So keep the vertex size minimum

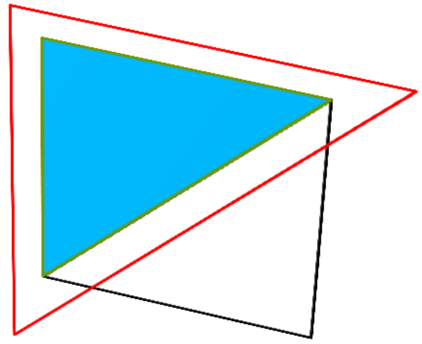


Edge

- The line is created by 2 vertices

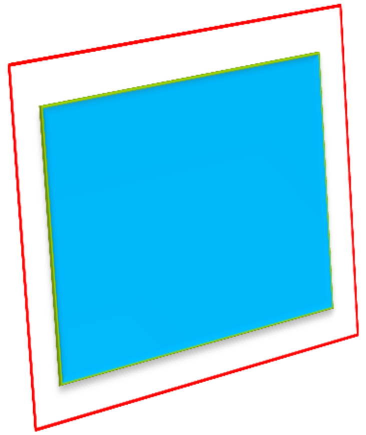


Introduction: Face



Triangle

- The intersection of the 3 edges defined by 3 vertices
- Basically define a triangle in 3D

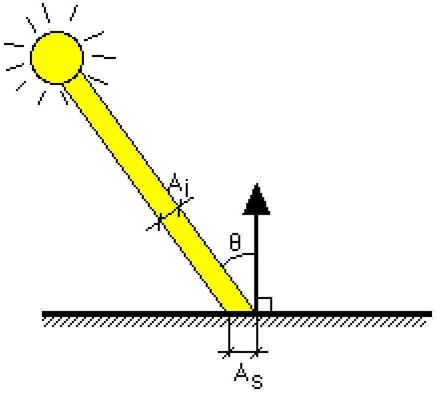


Polygon/Quad

- 4 vertex = 5 edges
- 2 faces/triangles

Introduction: Normal

- Are vectors perpendicular to a face

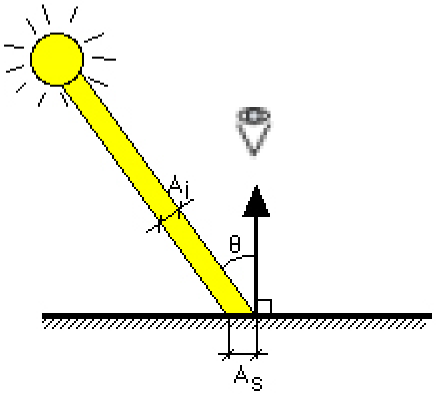


Picture 1

- See the picture 2:

<u>Question:</u>

Which case of lighting ray the eyes can see the illumination strongest?



Picture 2

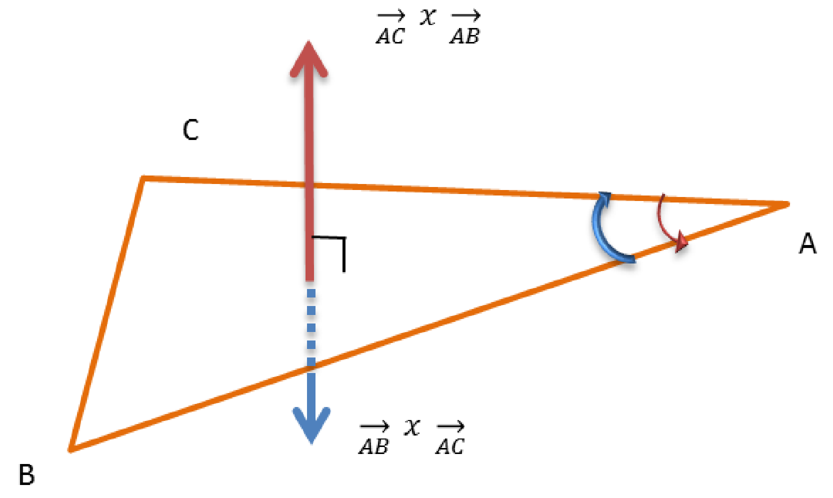
Introduction: Triangle points order

- The order of triangle points determine the direction of triangle normal.

Triangle normal = cross product $(\underset{AC}{\rightarrow}, \underset{AB}{\rightarrow})$

ACB = CBA = BAC = $\underset{AC}{\rightarrow} \times \underset{AB}{\rightarrow} = \underset{red}{\longrightarrow}$

ABC = $\underset{AB}{\rightarrow} \times \underset{AC}{\rightarrow} = \underset{blue}{\longrightarrow}$



Introduction: Pixel, Texel

Pixel

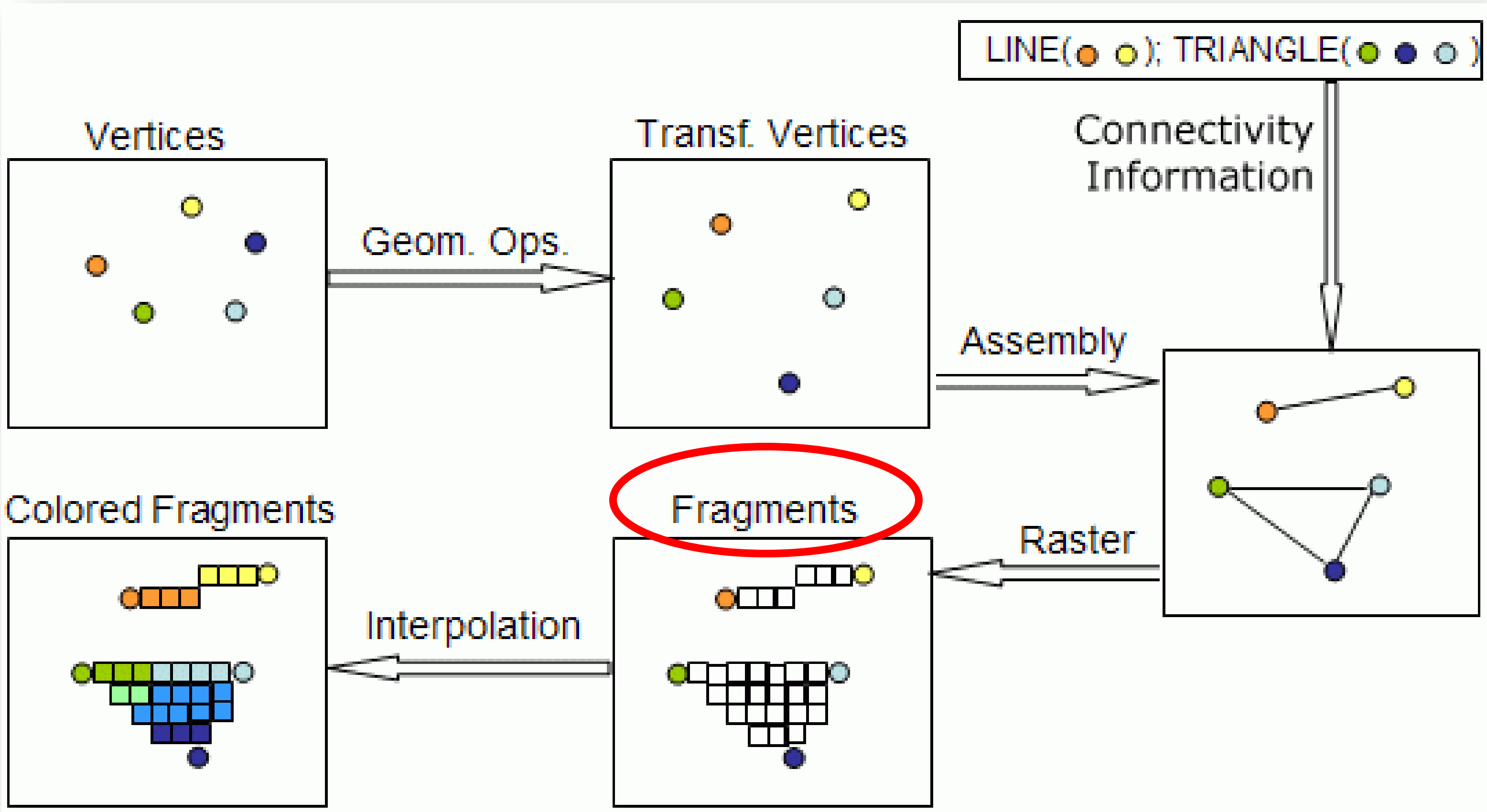
- Pixel is a point on the screen
- Is obtained through the combination of the three color channels, Red, Green and Blue

Texel

- Textures are represented by arrays of Texel, just as pictures are represented by arrays of pixels.
- A Texel is the pixel in a texture

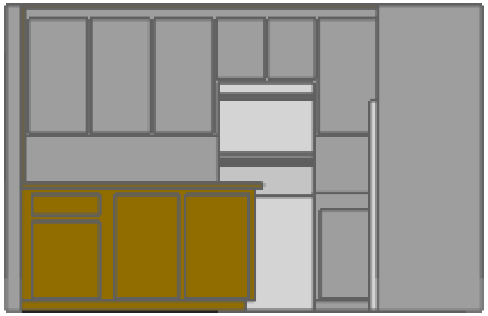
Introduction: Fragment

- Fragments are an intermediate between vertices and pixels.

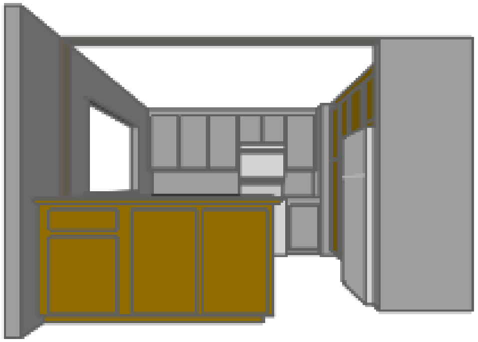


Introduction: Projection: Parallel & Perspective

- Represents the way a 3D space (3D world) is projected onto a 2D space (the screen)



Parallel projection



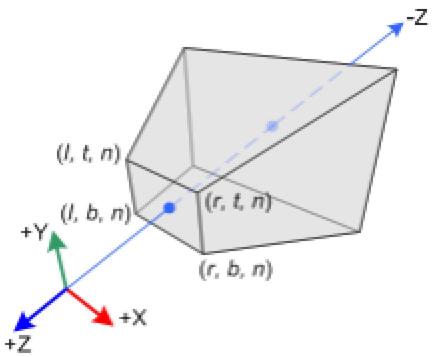
Perspective projection

- Parallel
  - Infinity eyes position
  - Projection lines (arrays) are parallel
  - Ortho is specific case of parallel projection when the ray is perpendicular to the projection plane
- Perspective
  - Smaller as their distance from the observer increases
  - Do not preserve the distance and size ratio

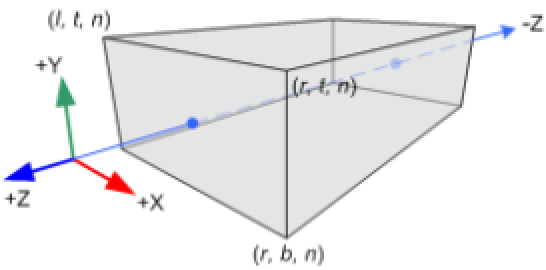
Introduction: Projection: Volume

- Limitted region for 3D rendering on device

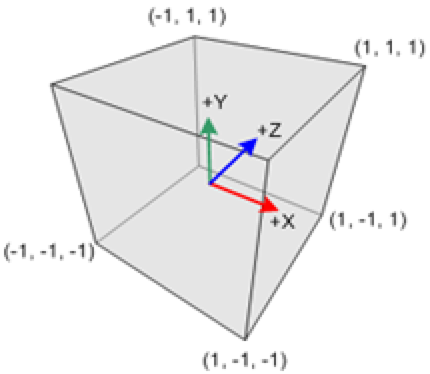
    Left, right, top, bottom, near, far



Perspective volume (Frustum)



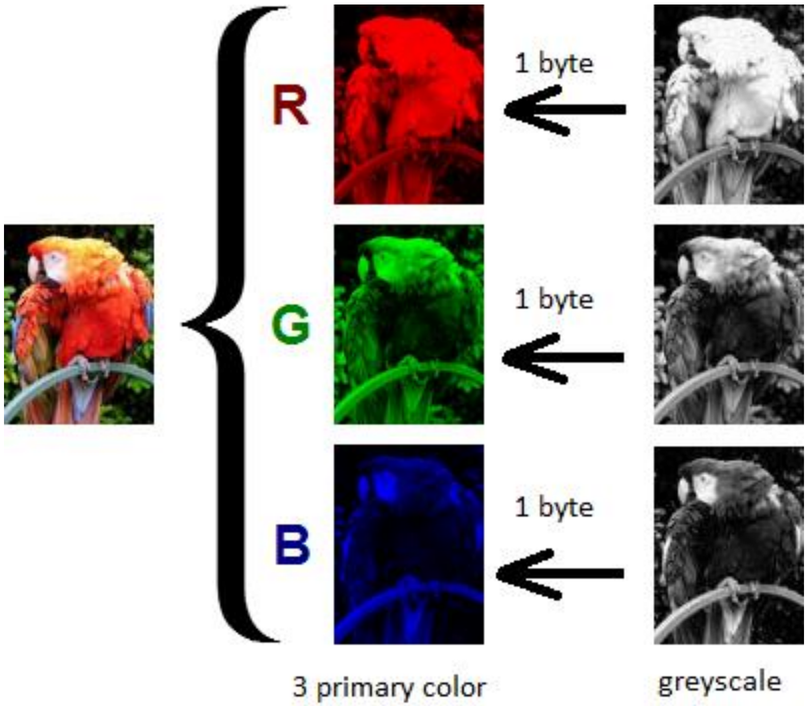
Orthographic volume



NDC

Introduction: Color channel

- Each pixel is made of combinations of primary color called Color Channel included: Red, Green, Blue, Alpha (RGBA) (A is optional)
- Each color channel stores color information valued 1byte [0, 255]
- Color value in GLSL ranging from [0.0, 1.0]
- More higher value of A channel, pixel is more opaque



Content

Introduction

Rendering pipeline

Shader

Basic GLSL-ES

Basic Math

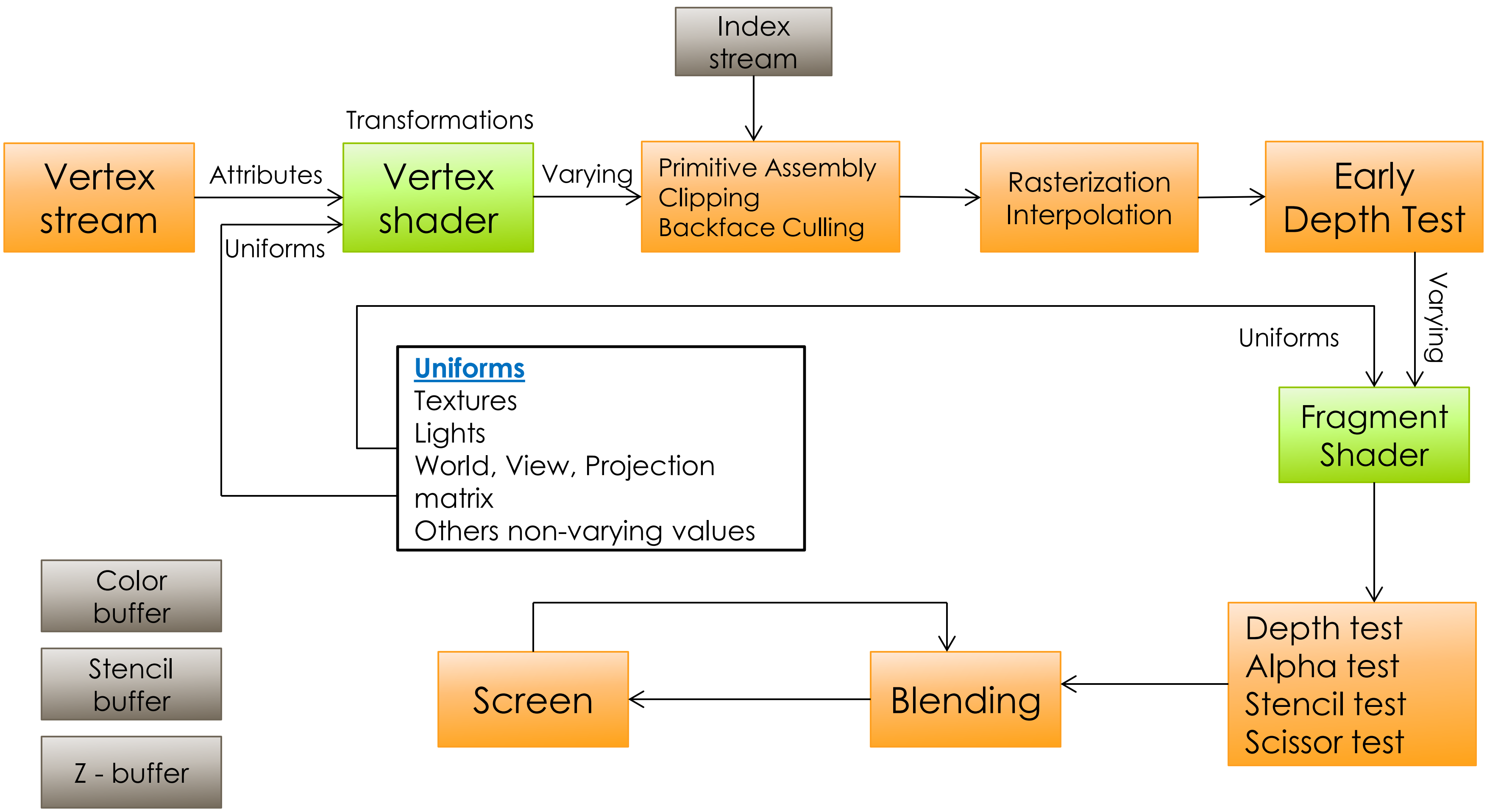
MVP matrices

Textures

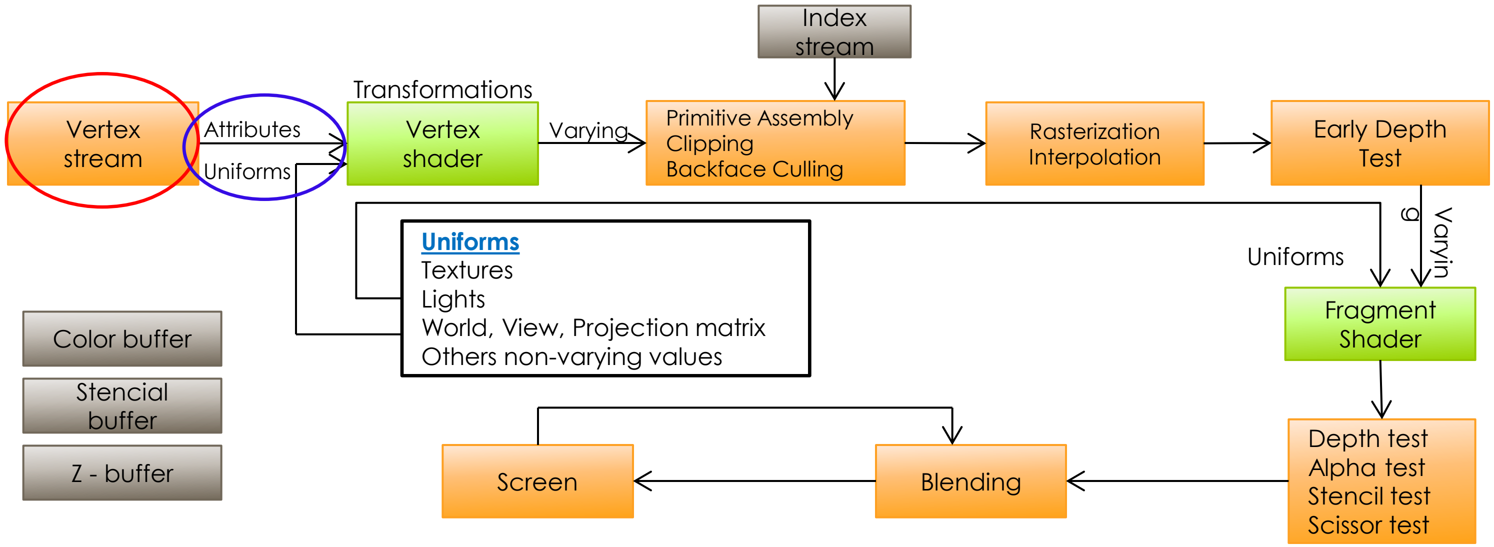
Obj model

Shader effect: Skydome using cube mapping

Rendering pipeline



Rendering Pipeline: Vertex Stream

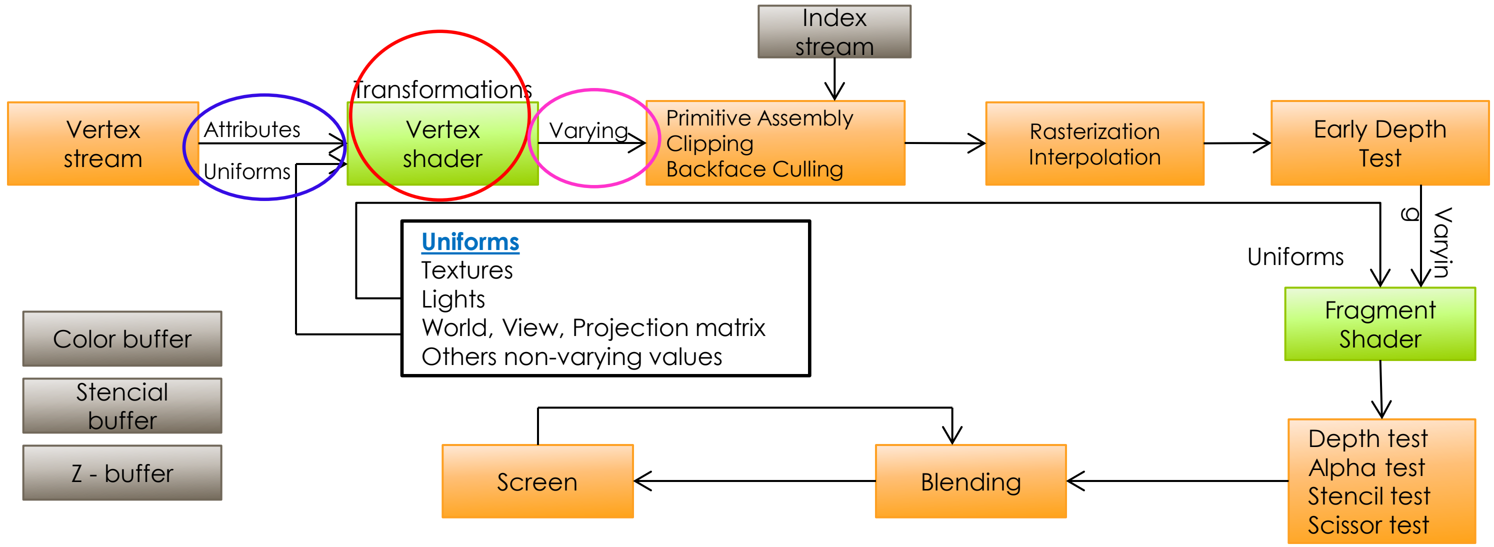


- Array of vertices sent from the RAM to the GPU memory
- Every vertex value is viewed as a attribute

✓ position
✓ Normal
✓ UV
✓ Color

of a vertex are all attributes

Rendering Pipeline: Vertex Shader

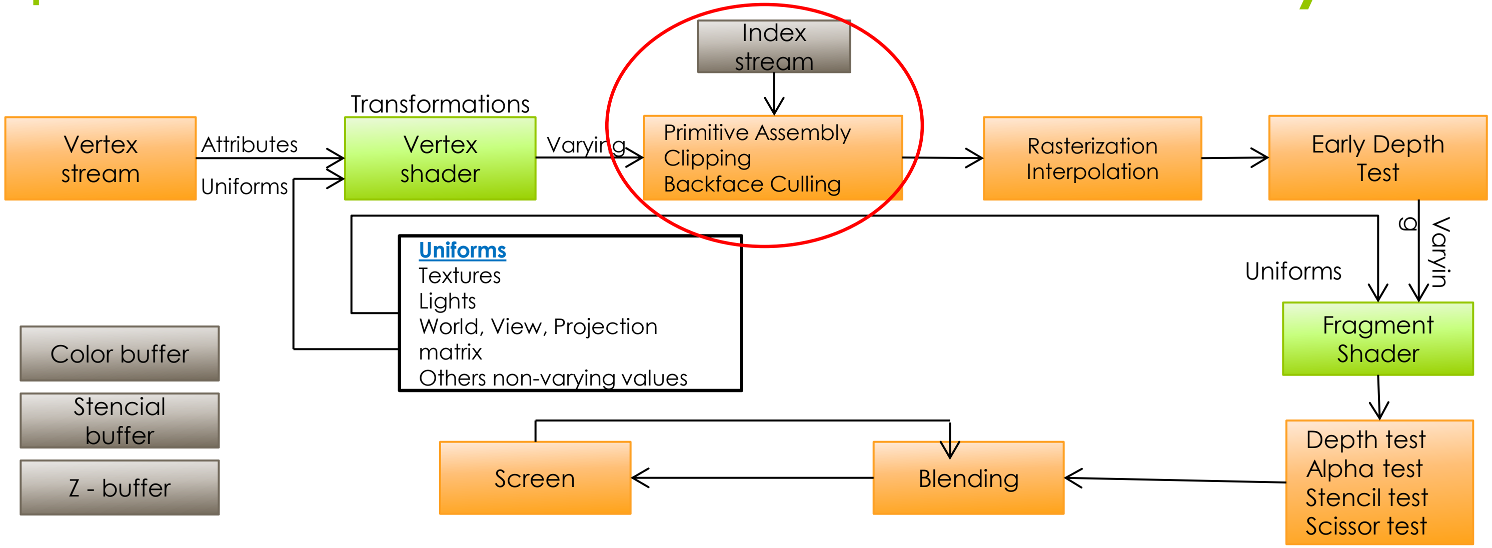


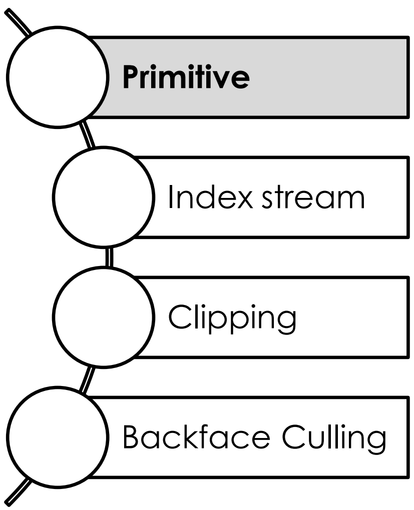
- Allow programmers define certain transformation on every vertex which enters the pipeline.
- Uniform: values that remain unchanged throughout the steps of the pipeline

Ex: Object's world, view and projection matrix

- Vertex Shader take in Uniform + attributes then pass out varying values for Fragment Shader

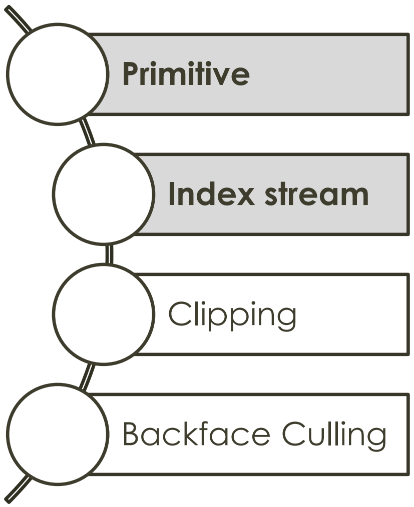
Rendering Pipeline: Index Stream and Primitive Assembly

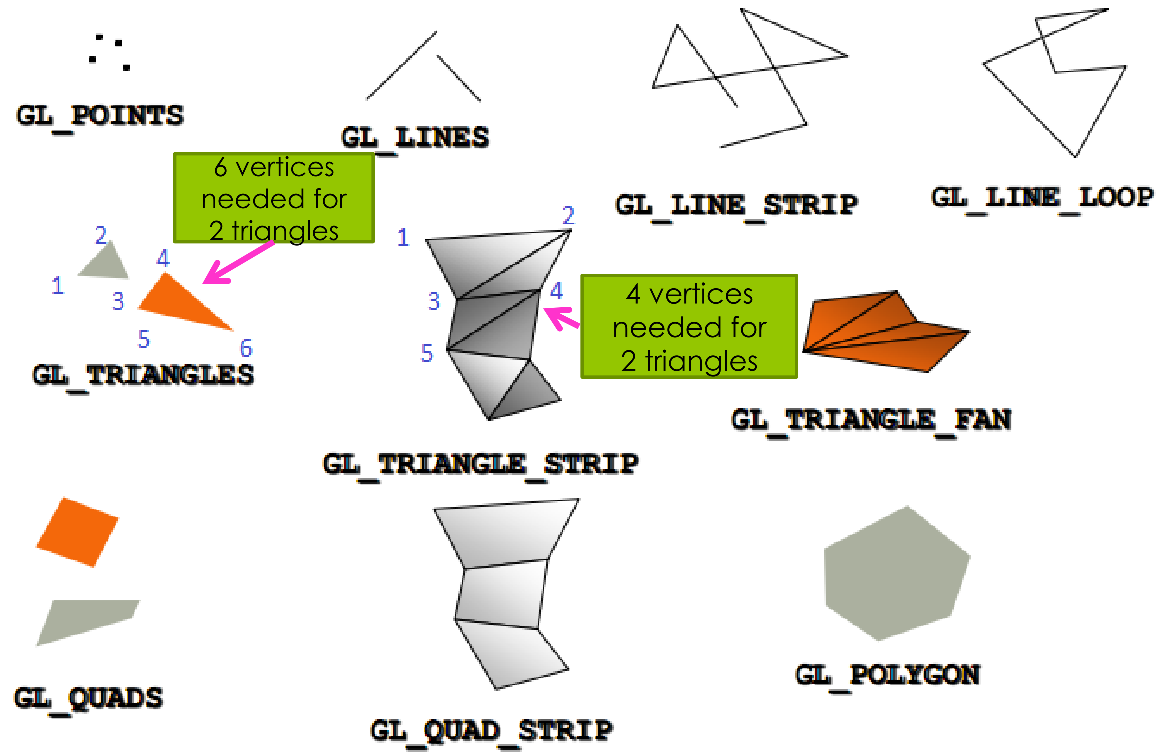




- Every object is broken down into basic geometry elements called primitive:
  - TRIANGLE
  - TRIANGLE_STRIP
  - QUAD
  - QUAD_STRIP
  - LINE,…
- The common use are the TRIANGLE and TRIANGLE_STRIP

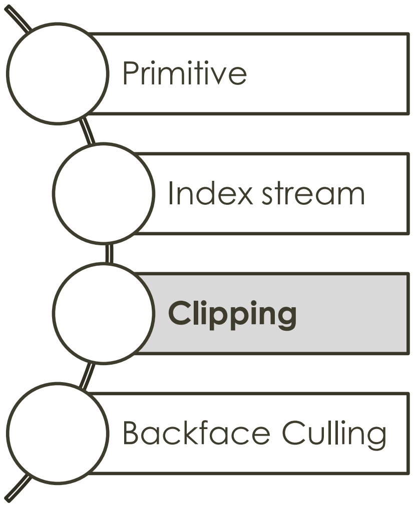
Rendering Pipeline: Index Stream and Primitive Assembly



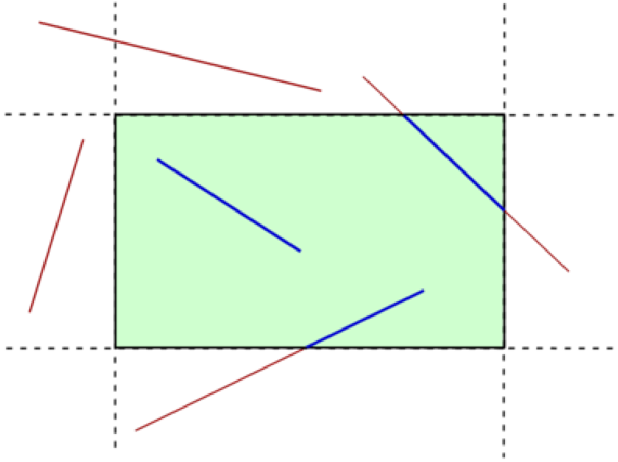


- ✓ What is index stream?
- ✓ To reduce traffic between RAM and GPU memory, using `glDrawElements()` instead of `glDrawArray()`

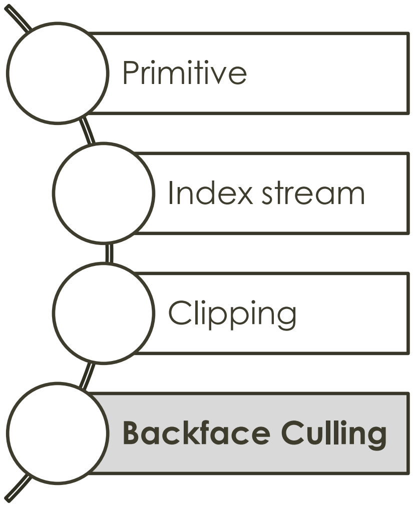
Rendering Pipeline: Index Stream and Primitive Assembly



- Passes all the primitives that are entirely inside the view volume
- Removes all primitives that are entirely outside the view volume
- Primitives that are partially inside the view volume require clipping: replaced by a new vertex located at the intersection (blue lines)



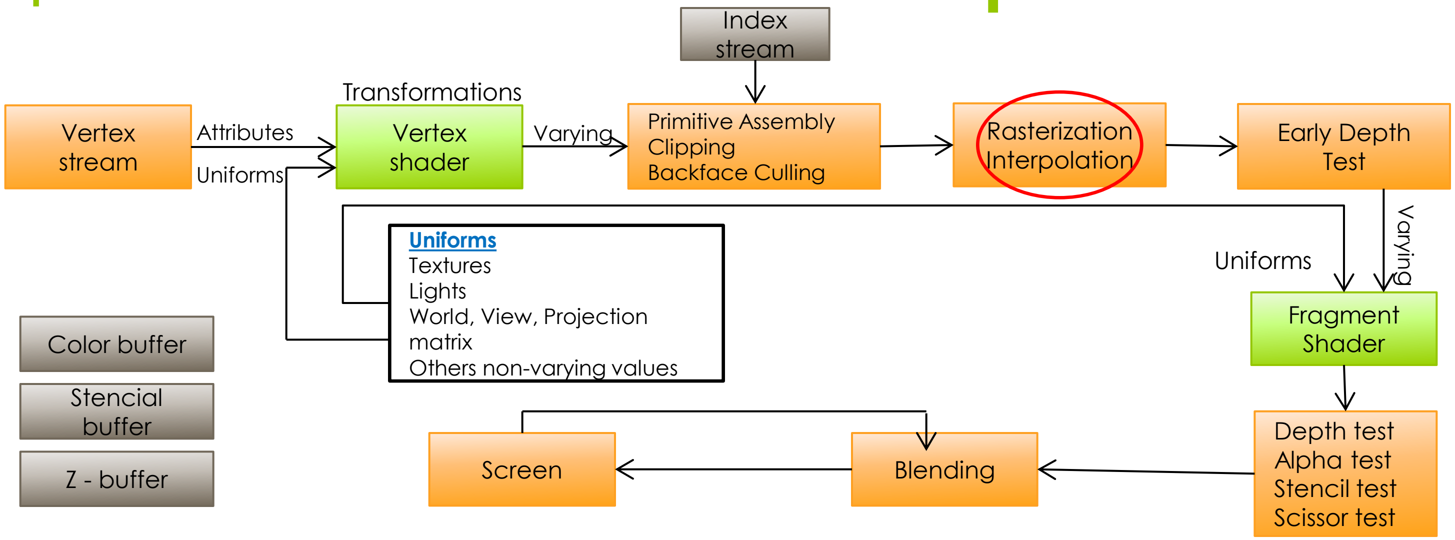
Rendering Pipeline: Index Stream and Primitive Assembly



- The backface culling (if is enabled):
  - It removes all primitives whose normals make an obtuse angle with the view vector.

- Number of fragments generated are greatly reduced → improving performance.

Rendering Pipeline: Rasterization and Interpolation



- **The rasterization**
  - The rasterization is the process in which the GPU determines the pixels covered by the primitive.

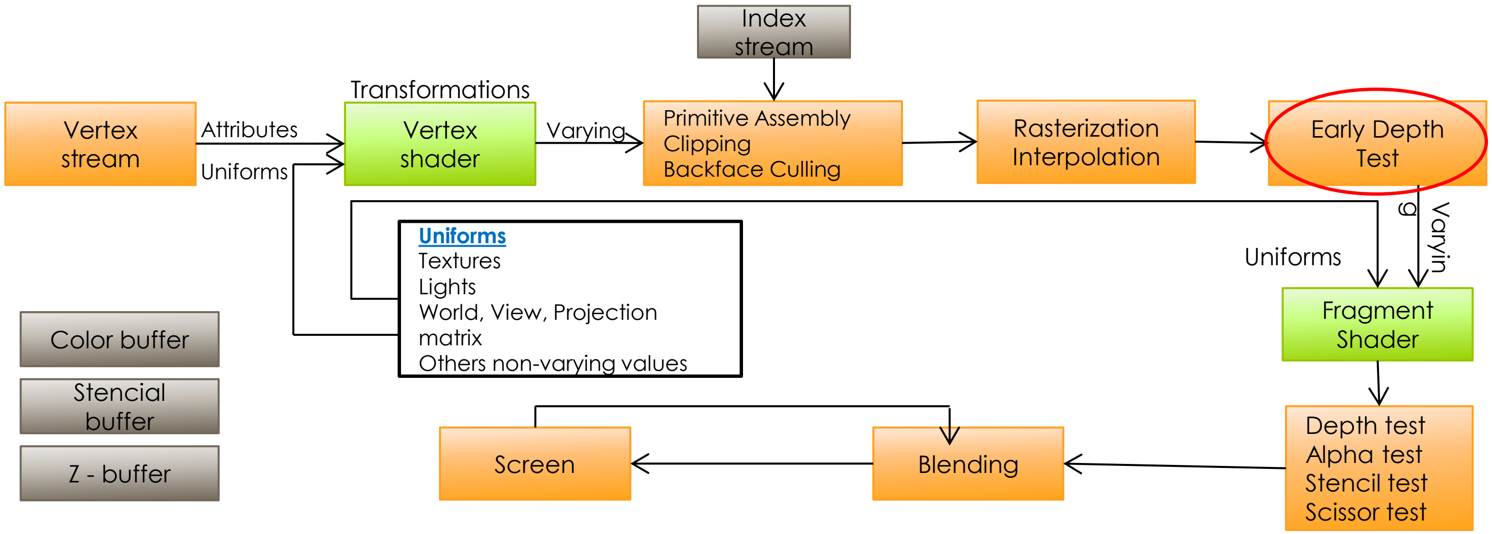
    The primitive can be filled or not!
- **The interpolation**
  - Takes place on all values coming from the vertex shader.
  - Converts primitives into a set of two-dimensional fragments, which are processed by the fragment shader

Rendering Pipeline: Rasterization and Interpolation (conts)

The interpolation (conts)

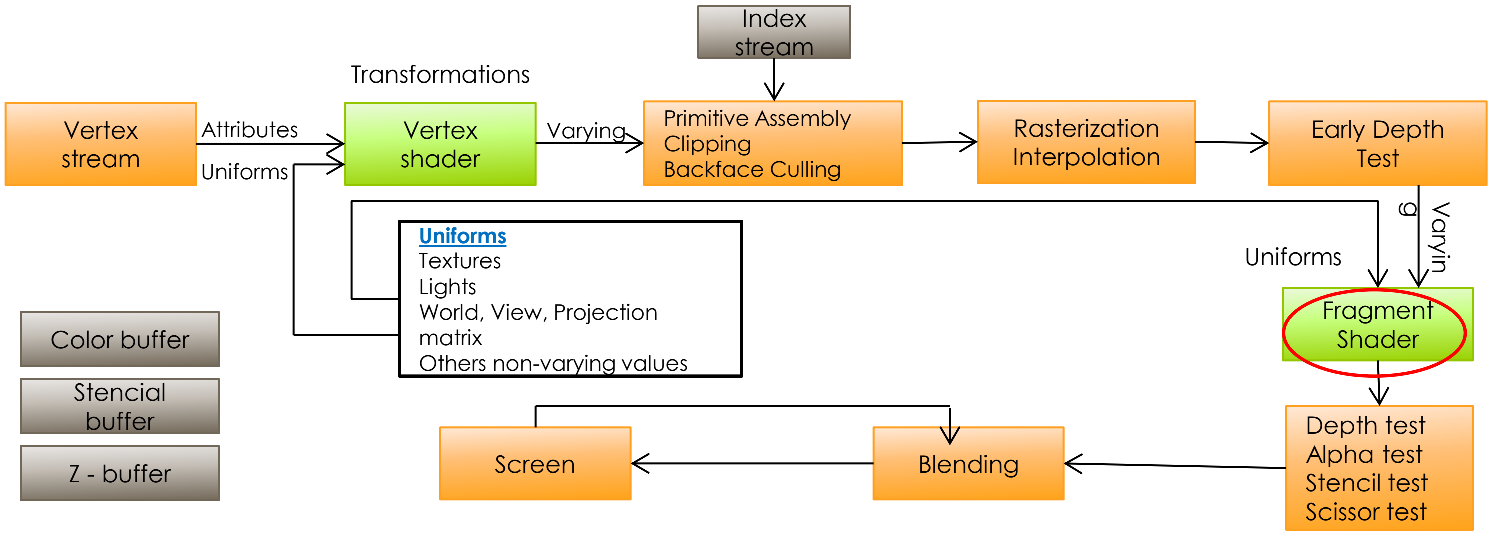
- Calculate the varying of each fragment → shouldn't use too many varying for optimization
- Discard automatically unused variables
  - Unused attribute will be passed with a default value (if don't enable the vertex attribute array)
  - Unused uniform won't be assigned an location (equals to -1).
  - Unused varying will not be computed / removed (depend on the implement of GPU)

Rendering Pipeline: Early Depth Test vs Depth Test



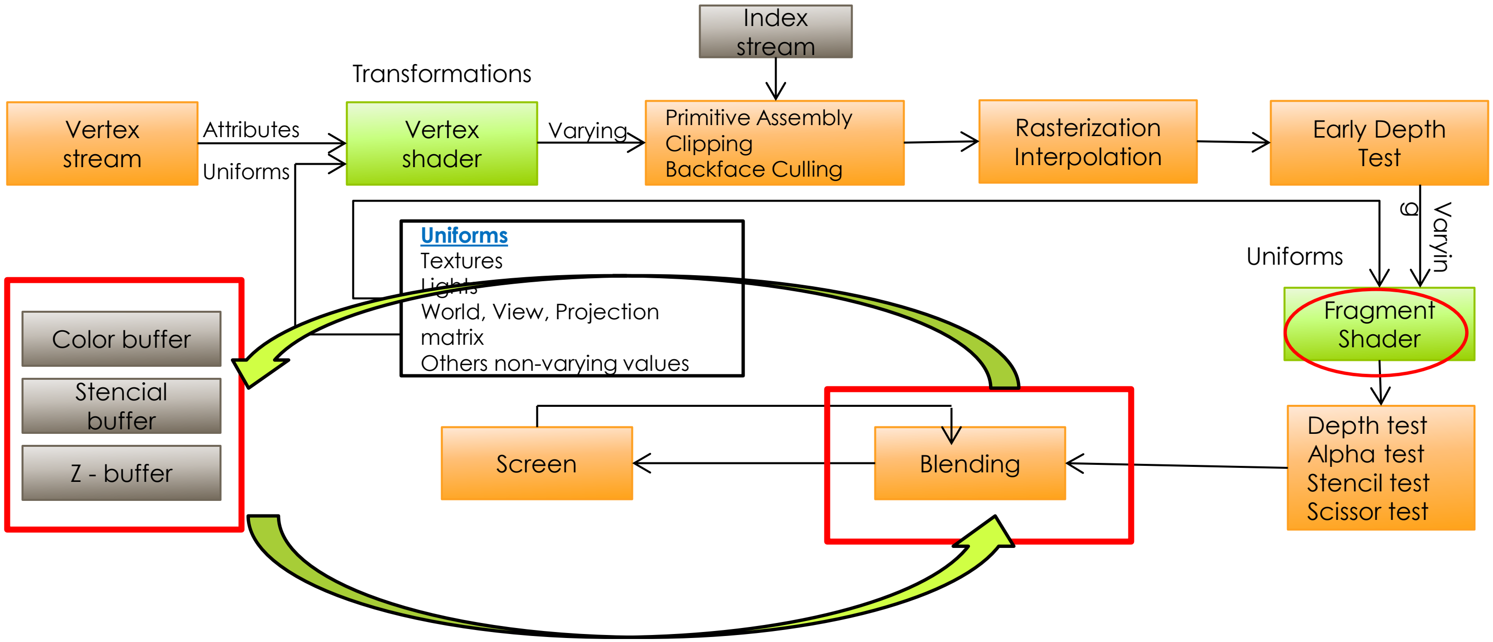
- Narrows down the number of fragments generated
- Dropping fragments which are drawn behind previously drawn fragment
- Referred as Z culling
- In Pipeline only Early Depth Test or Depth Test will be processed
- Early Depth Test only processes in some GPU as Adreno, Mali

Rendering Pipeline: Fragment Shader



- Allows the programmer to manipulate the color of fragments
- Out put is the color of the fragment.
- Make the alpha of all fragments oscillate between 0 and 1

Rendering Pipeline

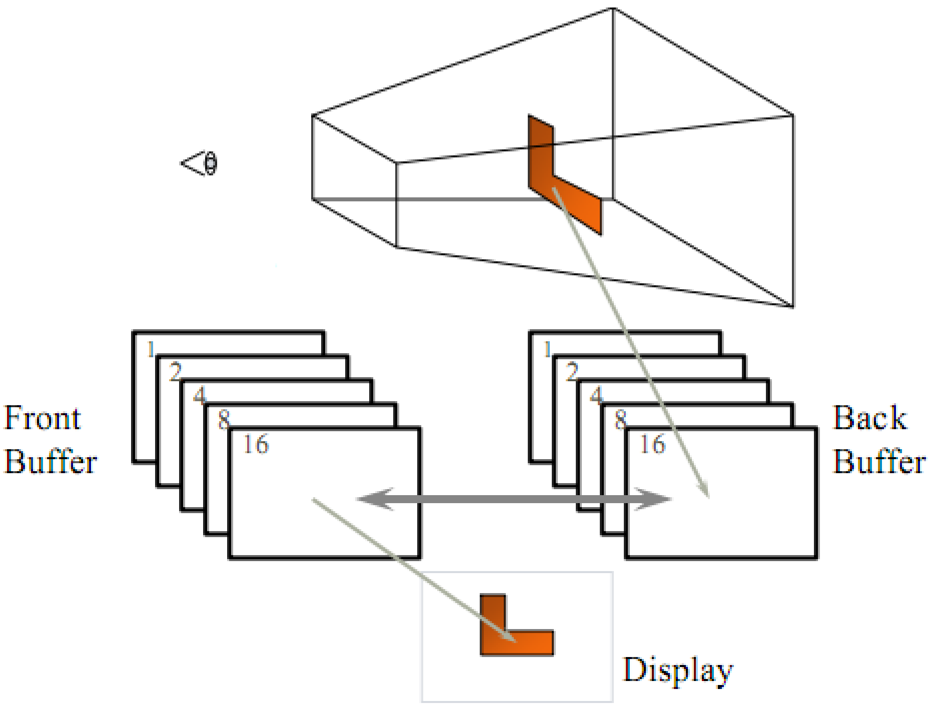


- Before go in Testing steps, you have to know about buffer which testing steps are done on it.

Rendering Pipeline: Rendering buffers

The color buffer (pixel buffer)

- Content value of color for each pixel
- Divide into two half (double buffer)
  - Front buffer
  - Back buffer



- Clear current color buffer
  `glClear(GL_COLOR_BUFFER_BIT)`

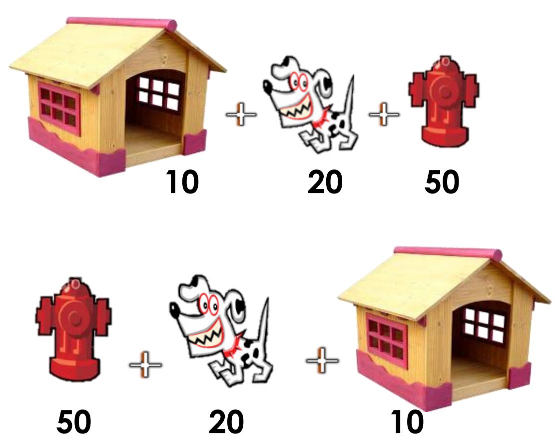
Rendering buffers

The stencil buffer

- An extra buffer at the programmer's disposal , and it stores an integer value for each pixel on the screen.
- Size: *sizeof(int) * screen_width * screen_height*.

The Z buffer (depth buffer)

- Contains per-pixel floating-point data for the z depth of each pixel rendered
- Keeps only the z value of pixel which is closest to the camera
- Ranging of depth per pixel is [0.0, 1.0] (near plane is 0.0 and far plane is 1.0)
- Vary size value from 8 → 32 bit



How buffer works?

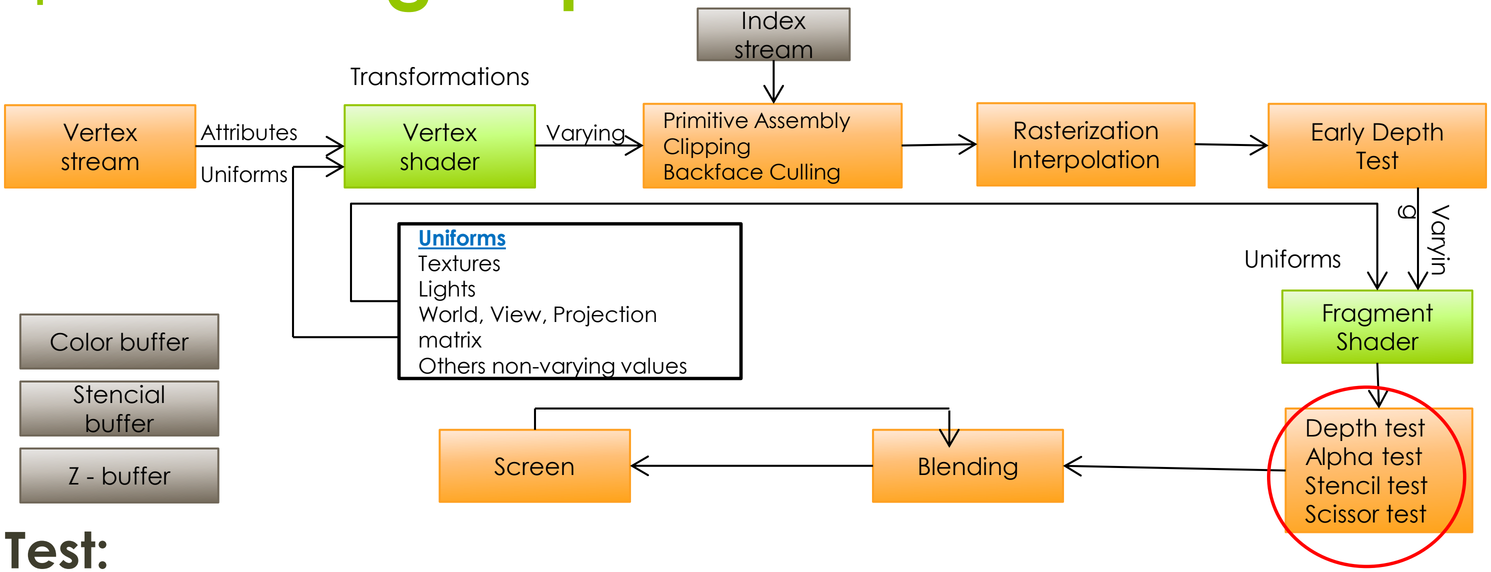
Rendering Pipeline: Rendering buffers

The Z fighting

- A phenomenon that occurs when two or more primitives have similar values in the z-buffer
- Higher value less chance to experience Z Fighting (16 bit is not as precise as 24 or 32 bit)
- How to remove z-fighting?

1) Use of a higher resolution depth buffer
2) Moving the polygons further apart
3) Reduce far plane

Rendering Pipeline: Testing steps



- **Depth Test:**
  - Test every fragment's z with the current z buffer
  - `glEnable(GL_DEPTH_TEST)`
  - Alpha fragments won't be written z-values to depth buffer ?
  - Solid nodes → front to back ?
  - Transparent nodes → back to front?
  - Transparent nodes must be renders at the end after others nodes.
  - Use `glDepthMask(GL_FALSE)` to disable writing to Z-buffer

Minimizing the amount of overdraw

Rendering Pipeline: Testing steps

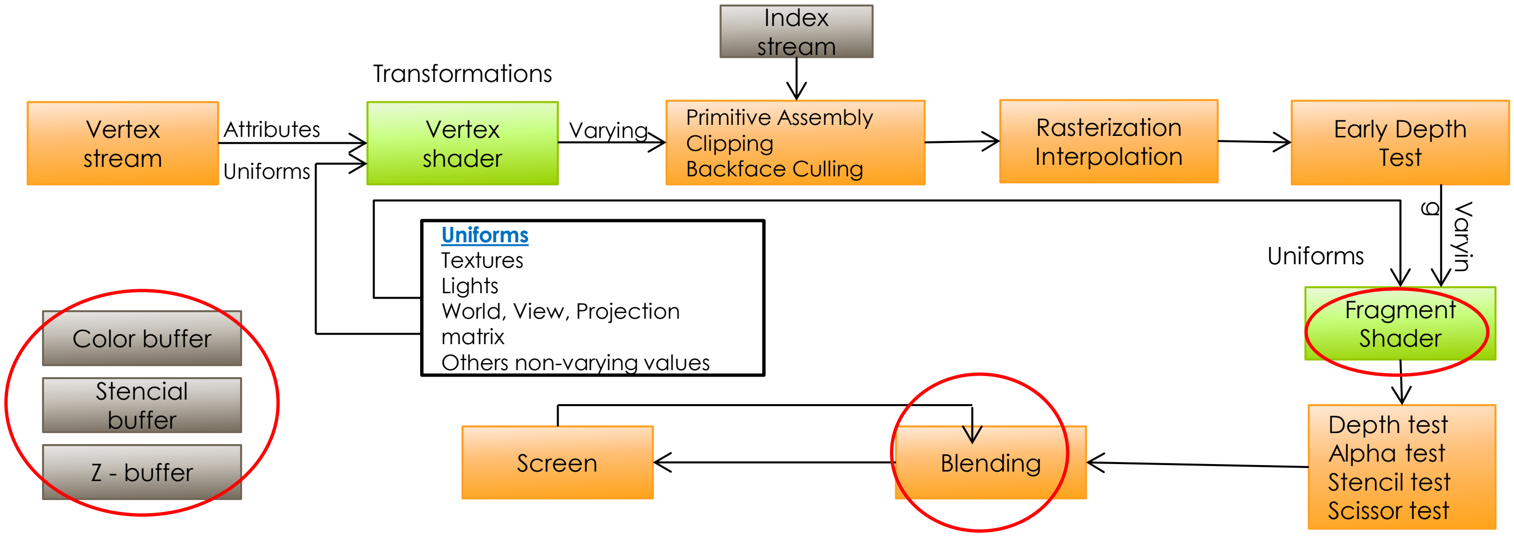
- **Alpha Test:**
  - Test each fragment's alpha channel with a value
  - Drop fragments which don’t meet the alpha requirements
  - Objects that make use of alpha test will be treated as opaque objects
  - The fragments that failed the alpha test won't update the Z Buffer

Rendering Pipeline: Testing steps

- **Stencil Test**
  - Test each fragment's coordinates with the stencil buffer
  - The stencil test + depth test → creating shadows, outlines and other graphical effects.
  - Besides, Stencil Test can be used to implement selection
  - `glEnable(STENCIL_TEST)`
- **Scissor Test**
  - Basically restricts the drawing area, creating viewports or rendering specific area
  - The order in which the tests are performed:

    Scissors Test → Alpha Test → Stencil Test → Depth Test
  - `glEnable(SCISSOR_TEST)`

Rendering Pipeline: Blending step



- Transparency effect is generated
- Computing the final color of the fragment
- After the blend step, the fragment becomes a pixel and it's color value is stored in the color buffer

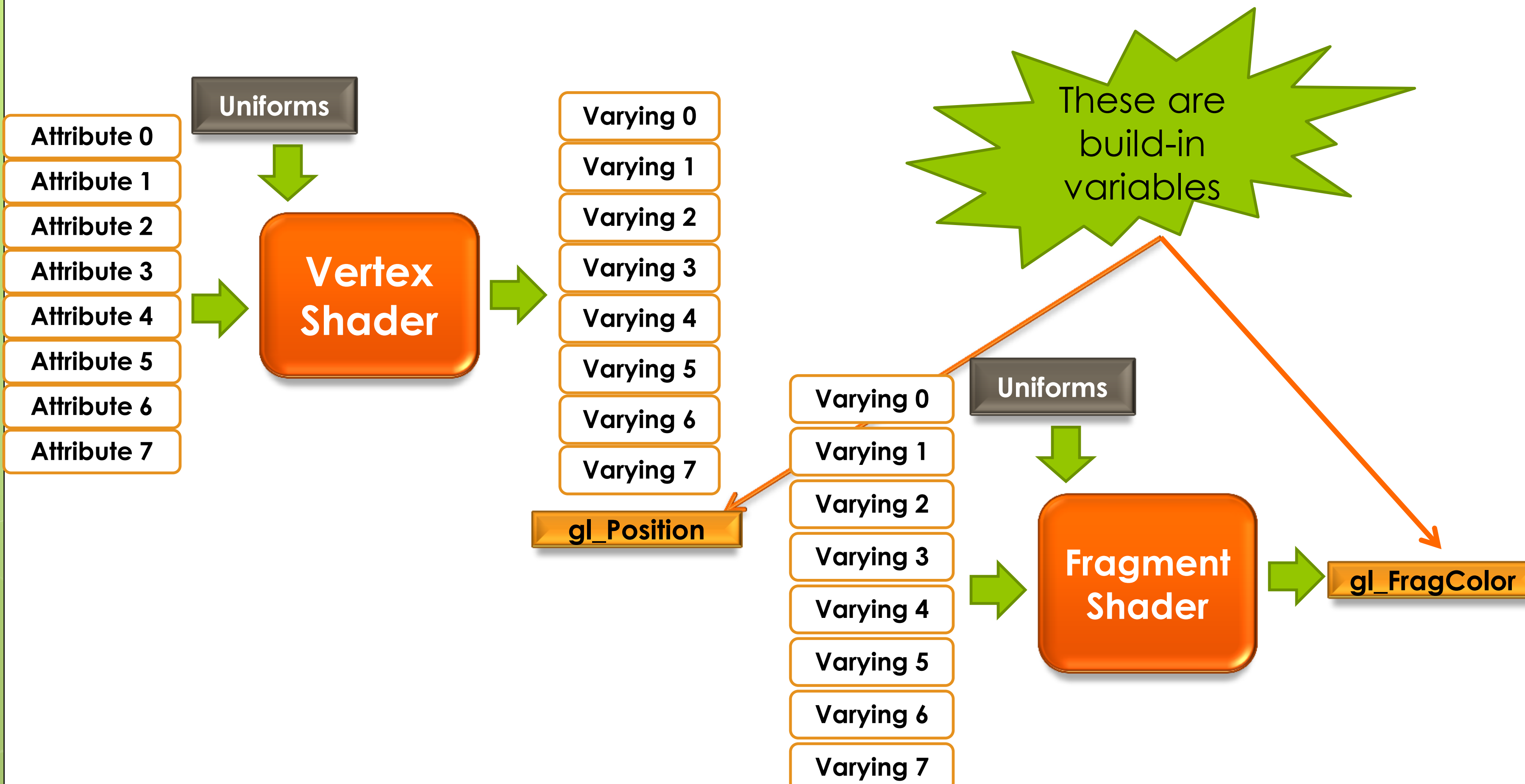
Content

- Introduction
- Rendering pipeline
- **Shader**
- Basic GLSL-ES
- Basic Math
- MVP matrices
- Textures
- Obj model
- Shader effect: Skydome using cube mapping

Shader

- Shader defines certain behavior for all vertices and fragments that come into the rendering pipeline
- Two types
  - Vertex shader
  - Fragment shader
- Following steps must be done before shaders can be used:
  - Compile shaders
  - Attach shaders to program
  - Link program
  - Use program to send necessary information to GPU

Shader



Uniforms
Attribute 0
Attribute 1
Attribute 2
Attribute 3
Attribute 4
Attribute 5
Attribute 6
Attribute 7
Vertex Shader
Varying 0
Varying 1
Varying 2
Varying 3
Varying 4
Varying 5
Varying 6
Varying 7
gl_Position
These are build-in variables
Varying 0
Varying 1
Varying 2
Varying 3
Varying 4
Varying 5
Varying 6
Varying 7
Uniforms
Fragment Shader
gl_FragColor

Shader

These variables are never used

When shaders are compiled, the varying, attributes, uniforms or any other unused local variable will be removed

- Pseudo:

Vertex shader	Fragment shader

uniform mat4 u_mvpMatrix;
attribute vec4 a_position;
attribute vec4 a_color;
varying vec4 v_color;
 	
void main()
{
//gl_Position must be set on
//every vertex shader
gl_Position = u_mvpMatrix *
a_position;
v_color = a_color;
}
	precision lowp float;
	varying vec4 v_color;
	
	void main()
	{
	//gl_FragColor must be set on
	//every vertex shader
	
	//gl_FragColor = v_color ;
	gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
	}

Why need precision?

Shader

Pseudo: LoadShader

```
GLuint LoadShader (Glenum type, const char* shadersrc)
{
		GLint compiled;
		// create a shader and get his handle.
		// The type can be either GL_VERTEX_SHADER or GL_FRAGMENT_SHADER
		GLuint aShader = glCreateShader (type);
		// send the shader source code to be compiled - shadersrc is of type char*
		glShaderSource(aShader, 1 , &shadersrc,NULL);
		// compile the shader
		glCompileShader(aShader);
		//check if the compilation of the shader succeeded
		glGetShaderiv(aShader,GL_COMPILE_STATUS,&compiled);
		//if compilation has failed delete the shader and return a null handle
		if (!compiled)
		{
				glDeleteShader(aShader);
				return 0;
		}
		return aShader;
}
```

Shader

Pseudo: LoadShader debug log

```
//debug by out put the info

glGetShaderiv ( shader, GL_INFO_LOG_LENGTH, &infoLen );

if ( infoLen > 1 )
{
        char* infoLog = (char *)malloc (sizeof(char) * infoLen );

        glGetShaderInfoLog ( shader, infoLen, NULL, infoLog );
        esLogMessage ( "Error compiling shader:\n%s\n", infoLog );

        free ( infoLog );
}
```

Shader

- Need a program to know what shaders to use
- A program is composed from a vertex shader and a fragment shader

Pseudo: Create Program

```
GLuint CreateProgram (const char* VertexShaderSrc, const char* FragmentShaderSrc)
{
    GLuint vertexShader;
    GLuint fragmentShader;
    GLuint programHandle;
    Glint linked;
    //load and compile the vertex shader
    vertexShader=LoadShader(GL_VERTEX_SHADER,VertexShaderSrc);
    //load and compile the fragment shader
    fragmentShader=LoadShader(GL_FRAGMENT_SHADER,FragmentShaderSrc);

    //create a program and get it's handle
    programHandle=glCreateProgram();
```

Shader

Pseudo: Create Program

```
    //attach the shaders to the program
    glAttachShader(programHandle,vertexShader);
    glAttachShader(programHandle,fragmentShader);

    //link the program
    glLinkProgram(programHandle);

    //check if linking was successful
    glGetProgramiv(programHandle,GL_LINK_STATUS,&linked);

    //delete the program if the link was not successful and return an invalid handle
    if (!linked)
    {
        glDeleteProgram(programHandle);
        return 0;
    }
    return programHandle;
}
```

Shader

- The only data we can send to the GPU RAM are **attributes** and **uniforms**.
- **Varying** values are passed from the vertex shader to the fragment shader
- Finding the location of the attributes and uniforms we want to send to the GPU by
  - `glGetAttribLocation`
  - `glGetUniformLocation`
- Make sure that these calls are AFTER the program has been successfully linked.

Shader

- For our two shaders we will have the following code:

```
GLint position = glGetAttribLocation(programHandle,"a_position");
GLint color    = glGetAttribLocation(programHandle,"a_color");// this return -1
GLint mvpMatrix = glGetUniformLocation(programHandle,"u_mvpMatrix");
```

- Use `glVertexAttribPointer` for each data sent to GPU
- Use `glUniformMatrix4fv` to send uniform matrix to GPU

```
glVertexAttribPointer(position, 3, GL_FLOAT,GL_FALSE,0,&vertices_positions);
if (color >= 0)
     glVertexAttribPointer(color, 4, GL_FLOAT,GL_FALSE,0, &vertices_color);

float afIdentity[16] = {1.0f, 0.0f, 0.0f, 0.0f,
                        0.0f, 1.0f, 0.0f, 0.0f,
                        0.0f, 0.0f, 1.0f, 0.0f,
                        0.0f, 0.0f, 0.0f, 1.0f};
glUniformMatrix4fv(mvpMatrix , 1, GL_FALSE, afIdentity);
```

- Notice: `glGetAttribLocation` will return -1 for `a_color` because after the compile it has been removed.

Shader

```
void glVertexAttribPointer(
                GLuint  index,
                GLint  size,
                GLenum  type,
                GLboolean  normalized,
                GLsizei  stride,
                const GLvoid *  pointer);
```

```
void glUniformMatrix4fv(
        GLint  location,
        GLsizei  count,
        GLboolean  transpose,
        const GLfloat * value);
```

- *Position* inside the shader of the attribute
- *Size* of the attribute
- *Type* of the attribute
- The *stride* (the offset in the vector between the data)
- A *pointer* to the buffer that holds the values
- Stride is useful when you concatenate data.
- An arrays of vertices, with each vertex being defined by a structure of floats

```
struct Vertex
{
    float x,y,z;
    float u,v;
    float r,g,b,a;
}
```

- *Location*: Specifies the location of the uniform value to be modified.
- *Count*: Specifies the number of matrices that are to be modified
- *Transpose*: Specifies whether to transpose
- *Value*: Specifies a pointer to an array

Shader

```
Vertex* verticesArray = LoadObjectFromFile("aObject");
GLfloat* ptr  = (GLfloat *)verticesArray;

glVertexAttribPointer(position, 3, GL_FLOAT, sizeof(Vertex), ptr);
glVertexAttribPointer(color,     4, GL_FLOAT, sizeof(Vertex), ptr+5);

glDrawArrays(GL_TRIANGLES, 0, num_of_face * 3);

glEnableVertexAttribArray(position);
glEnableVertexAttribArray(color); //if possible
```

What happens without this method?

Must call to attributes will actually get to the GPU

Shader

- Uniforms stay the same for each vertex that will be processed
- Sending uniforms use specific functions, depending on the uniform types.
- Ex:

```
float[3] red_color;
.....
glUniform1f(time, deltaT);
glUniform3fv(red, 1, &red_color);
```

One with 1 float

One with 3 float

Content

- Introduction
- Rendering pipeline
- Shader
- **Basic GLSL-ES**
- Basic Math
- MVP matrices
- Textures
- Obj model
- Shader effect: Skydome using cube mapping

Basic GLSL

- Introduction
- GLSL declaration
- References

Basic GLSL: Introduction

What Is GLSL? Open**GL** **S**hading **L**anguage

- C/C++ similar high level programming language for several parts of the graphic card
- Can code (right up to) short programs, called shaders, which are executed on the GPU.

Why Shaders?

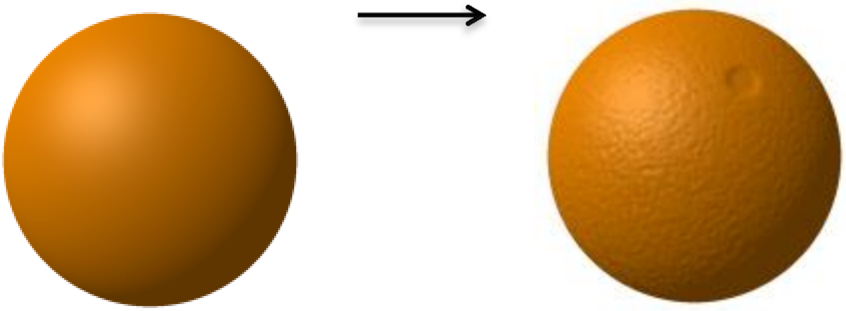
- In OpenGLES 1.1, the graphic pipeline could only be configured, but not be programmed.
- With Shaders you are able to program: your lighting mode, Shadows, Environment Mapping, Bump Mapping, Parallax Bump Mapping, HDR, and much more!

Basic GLSL: Introduction

Environment Mapping

Shadow

Bump Mapping



Basic GLSL: Introduction

- You have nearly full control over what is happening with each vertex / fragment

Vertex shader	Fragment shader
✓ Vertex Transformation ✓ Normal Transformation, Normalization and Rescaling ✓ Lighting ✓ Texture Coordinate Generation and Transformation ✓ And more.	✓ Texture access and application (Texture environments) ✓ Fog ✓ Lightning ✓ And more

Basic GLSL: GLSL declaration

- Data type
- Vector constructor
- Matrix constructor
- Array constructor
- Structure constructor
- Vector component
- Function definition

Basic GLSL: GLSL declaration

Data types

- There are four main types: float, int, bool and sampler.
- For the first three types, vector types are available:

vec2, vec3, vec4	2D, 3D and 4D floating point vector
ivec2, ivec3, ivec4	2D, 3D and 4D integer vector
bvec2, bvec3, bvec4	2D, 3D and 4D boolean vectors

- For floats here are also matrix types:

mat2, mat3, mat4	2x2, 3x3, 4x4 floating point matrix

- Some common errors (cause error without notices) :

```
float b = 3; // must be 3.0
```

Basic GLSL: GLSL declaration

Vector constructors

- vec3(float) // initializes each component of with the float
- vec2(float, float) // initializes a vec2 with 2 floats
- ivec3(int, int, int) // initializes an ivec3 with 3 ints
- vec3(vec4) // drops the fourth component of a vec4
- vec3(vec2, float) // vec3.x = vec2.x, vec3.y = vec2.y, vec3.z = float
- vec3(float, vec2) // vec3.x = float, vec3.y = vec2.x, vec3.z = vec2.y
- vec4(vec2, vec2) // ???

Examples

- vec4 color = vec4(0.0, 1.0, 0.0, 1.0); // 1.0 not 1.0f like C++
- vec4 rgba = vec4(1.0); // sets each component to 1.0
- vec3 rgb = vec3(color); // drop the 4th component

Basic GLSL: GLSL declaration

Matrix Constructors

- To initialize the diagonal of a matrix with all other elements set to zero:

  ```
  mat2(float), mat3(float), mat4(float)
  ```

- To initialize a matrix by specifying vectors or scalars, the components are assigned to the matrix elements in column-major order:

  ```
  mat2(vec2, vec2); // one column per argument
  mat3(vec3, vec3, vec3); // one column per argument
  mat4(vec4, vec4, vec4, vec4); // one column per argument
  mat3x2(vec2, vec2, vec2); // one column per argument
  ```

Basic GLSL: GLSL declaration

```
mat2(float, float, // first column
     float, float); // second column
mat3(float, float, float, // first column
     float, float, float, // second column
     float, float, float); // third column
mat4(float, float, float, float, // first column
     float, float, float, float, // second column
     float, float, float, float, // third column
     float, float, float, float); // fourth column
mat2x3(vec2, float, // first column
       vec2, float); // second column
mat2x3(mat4x2); // takes the upper-left 2x2 of the mat4x4, last row is 0,0
mat4x4(mat3x3); // puts the mat3x3 in the upper-left, sets the lower right component to 1, and the rest to 0
```

Basic GLSL: GLSL declaration

Array Constructors

- Array types can also be used as constructor names, which can then be used in expressions or initializers. For example:

```
const float c[3] = float[3](5.0, 7.2, 1.1);
const float d[3] = float[](5.0, 7.2, 1.1);
float g;
...
float a[5] = float[5](g, 1, g, 2.3, g);
float b[3];
b = float[3](g, g + 1.0, g + 2.0);
```

Basic GLSL: GLSL declaration

Structure Constructors

- Once a structure is defined, and its type is given a name, a constructor is available with the same name to construct instances of that structure. For example:

```
struct light
{
    float intensity;
    vec3 position;
};

light lightVar = light(3.0, vec3(1.0, 2.0, 3.0));
```

Basic GLSL: GLSL declaration

Vector Components

- The component names supported are:

  {x, y, z, w} Useful when accessing vectors that represent points or normals

  {r, g, b, a} Useful when accessing vectors that represent colors

  {s, t, p, q} Useful when accessing vectors that represent texture coordinates

- The component names *x, r,* and *s* are, for example, synonyms for the same (first) component in a vector.
- Accessing components beyond vector type is error. For example:

```
vec2 pos;
pos.x // is legal
pos.z // is illegal
```

Basic GLSL: Data Types In GLSL

Vector component (conts)

- The component selection syntax allows multiple components to be selected by appending their names (from the same name set) after the period (.).

```
vec4 v4;
v4.rgba; // is a vec4 and the same as just using v4,
v4.rgb; // is a vec3,
v4.b; // is a float,
v4.xy; // is a vec2,
v4.xgba; // is illegal - the component names do not come  from the
         // same set.
```

Basic GLSL: GLSL declaration

Vector component (conts)

- The order of the components can be different to swizzle them, or replicated:

```
vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);
vec4 swiz= pos.wzyx; // swiz = (4.0, 3.0, 2.0, 1.0)
vec4 dup = pos.xxyy; // dup = (1.0, 1.0, 2.0, 2.0)
```

- The component group notation can occur on the left hand side of an expression.

```
vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);
pos.wx = vec2(7.0, 8.0); // pos = (8.0, 2.0, 3.0, 7.0)
pos.xx = vec2(3.0, 4.0); // illegal - 'x' used twice
pos.xy = vec3(1.0, 2.0, 3.0); // illegal - mismatch between vec2 and vec3
//pos[2] refers to the third element of pos and is equivalent to pos.z.
```

Basic GLSL: GLSL declaration

Function Definitions

- A function is declared as the following example shows:

```
// prototype
returnType functionName (type0 arg0, type1 arg1, ..., typen argn);
```

- And a function is defined like

```
// definition
returnType functionName (type0 arg0, type1 arg1, ..., typen argn)
{
    // do some computation
    return returnValue;
}
```

Basic GLSL: GLSL declaration

Function Definitions (conts): Basic built-in functions

```
float dot (genType x, genType y)        //Returns the dot product of x and y
vec3 cross (vec3 x, vec3 y)             //Returns the cross product of x and y
genType normalize (genType x)           // Returns a vector in the same direction as x
                                                  but with a length of 1
genType reflect (genType I, genType N)//For the incident vector I and surface
                                       //orientation N, returns the reflection
                                       //direction. N must already be normalized
float length (genType x)                //Returns the length of vector X
genType mix (genType x,genType y,float a)      //Returns the linear blend of x and y
genType abs (genType x)                 //Returns x if x >= 0, otherwise it returns –x
genType sqrt (genType x)                //Returns square root of (x)
genType pow (genType x, genType y)//Returns x raised to the y power
genType sin (genType angle)             //The standard trigonometric sine function
```

References

- **Docs:**
  - Slide, demo and detail book about workshop content:

\\sai-data01\Documents\Specialized\Programming\Training\01. MegaTraining\Basic\3D & OpenGL\GLES 2.0 workshop

  - More about shader and opengles2.0:

\\sai-data01\Documents\Specialized\Programming\Training\01. MegaTraining\Basic\3D & OpenGL\OpenGL_ES

\\sai-data01\Documents\Specialized\Programming\Training\01. MegaTraining\Basic\3D & OpenGL\Shader

- **Kits:**

\\sai-data01\Documents\Specialized\Programming\Training\01. MegaTraining\Basic\3D & OpenGL\Kits

  - To load .obj file: GLC_Player_1.5.0-setup.exe
  - To compile shader source: OGLES2_WINDOWS_PCEMULATION_2.04.24.0811.msi

- **Model and others:**

\\sai-data01\Documents\Specialized\Programming\Training\01. MegaTraining\Basic\3D & OpenGL