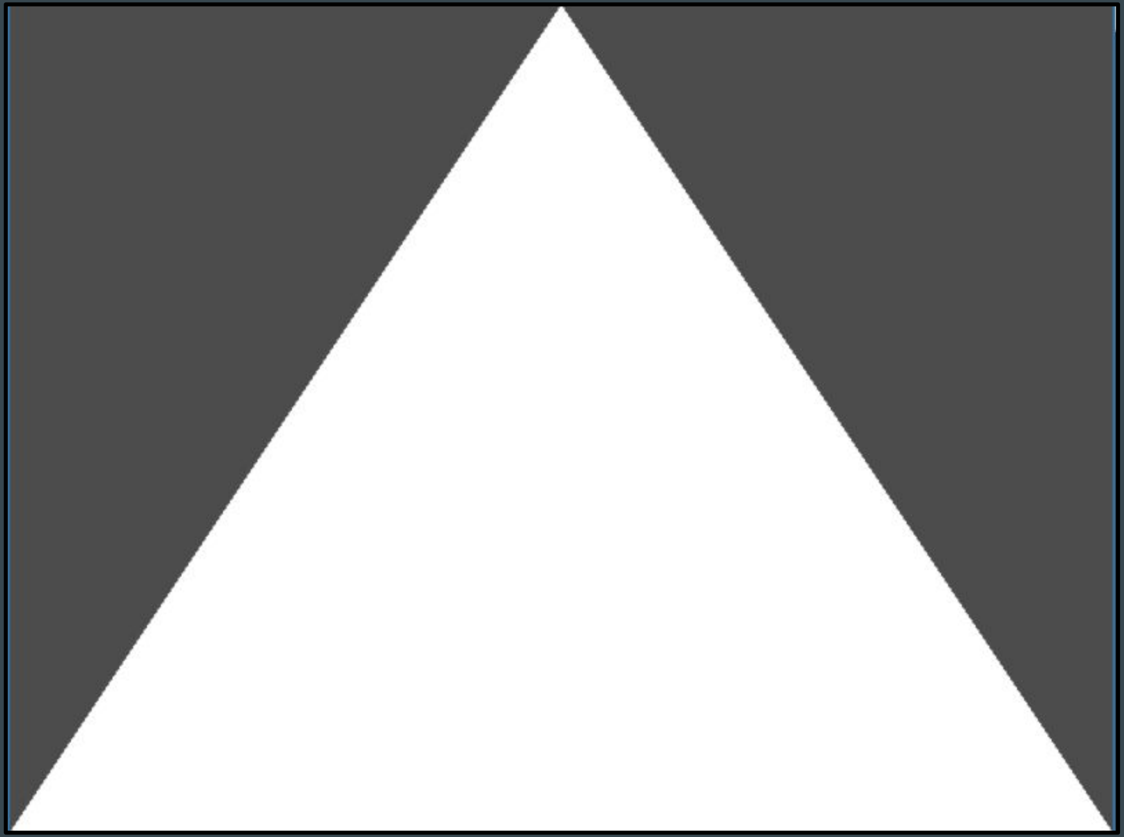
Drawing with OGL

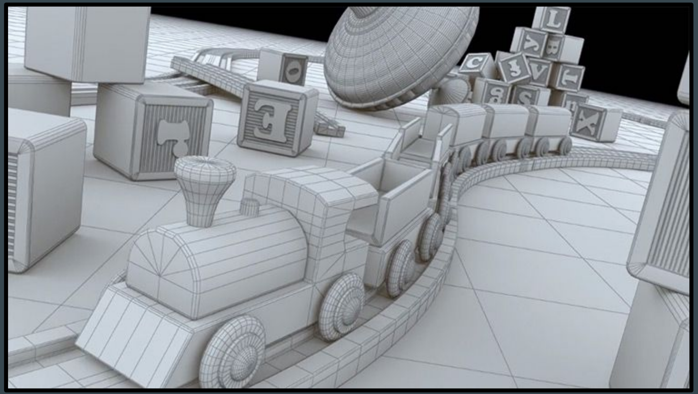
Carlos Fuentes

End Goal



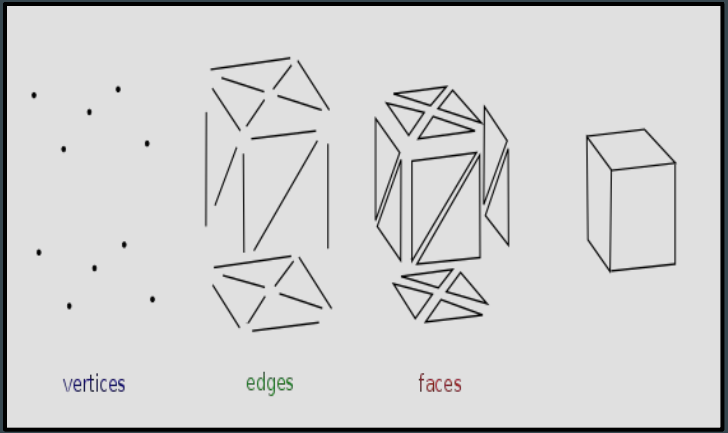
Polygon Mesh

- Models that are rendered by GPU are Polygon Meshes
- Actually GPUs are built to render Triangle Meshes



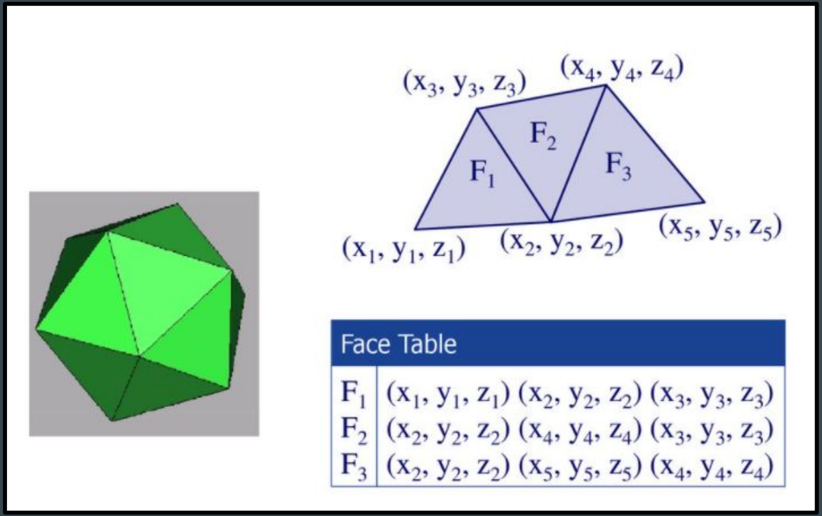
Polygon Mesh

- A polygon mesh is a collection of **vertices, edges** and **faces** that defines the shape of an object.
- Meshes may have different representations, but for rendering the most widely used are a **list of faces and a list of vertices**.



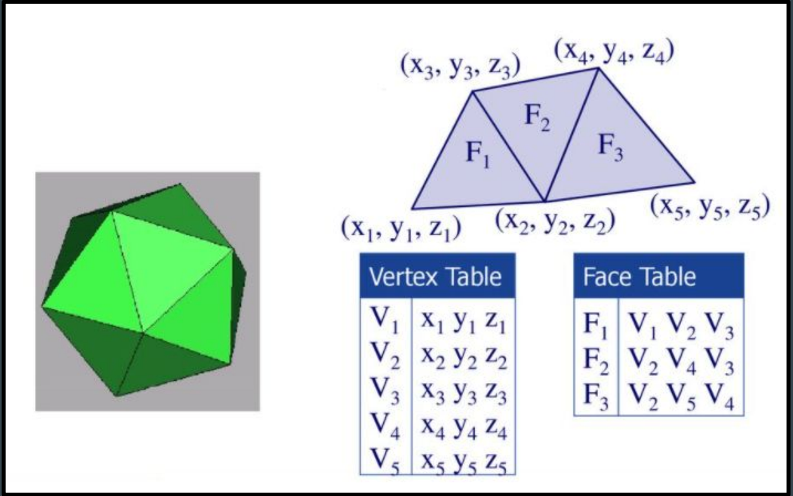
Polygon Mesh

- On possible representation of a polygon mesh is **specifying directly the vertices** that forms each face.
- This is a compact representation but **vertex are repeated**.



Polygon Mesh

- Another representation is to have a list of faces as **indexes** to a table of vertices.
- No repeating vertices is important specially because could contain several attributes:
  - (x,y,z) vertex **position**
  - (x,y,z) vertex **normal**
  - (rgba) vertex **color**



Send mesh to render: Immediate mode

- **Old OpenGL versions** had different approaches to sending vertices to GPU.
- **Immediate mode** was first approach. Consists on specifying each vertex through a function call with implicit face representation.
- Problem ⇒ Thousands of vertices ??? ⇒ Thousands of function calls!!!!

```
glBegin(GL_TRIANGLES); // Each 3 vertices are a new triangle
glVertex3f(1.0f, 2.0f, 3.0f);
glVertex3f(4.0f, 2.0f, 1.0f);
glVertex3f(6.0f, 1.0f, 3.0f);
glEnd();
```

Send mesh to render : regular vertex arrays

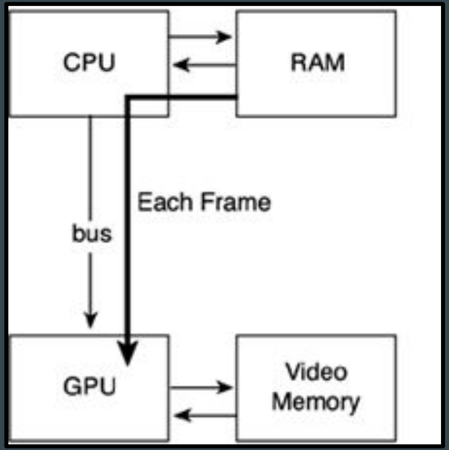
- For avoiding **function call overhead** depending on the number of vertices, **vertex arrays** were invented

```
int numfaces = 64;
float *vtx = new float[numfaces*9];
// fill vertex array at loading
...
...

// send vertex array to draw
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT,0, &vtx[0]);
glDrawArrays(GL_TRIANGLES, 0, 3*numfaces);
```

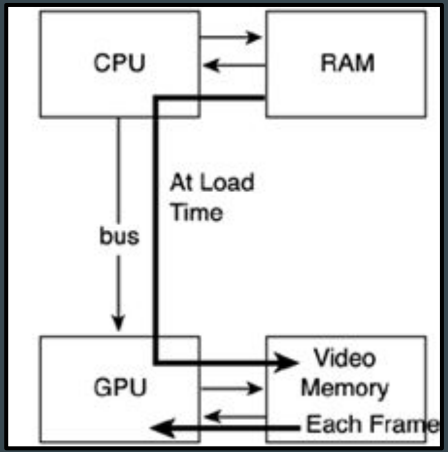
Send a Mesh to render: Bus limitation

- Vertex arrays had to send each frame the vertices from CPU to GPU.
- Number of vertices rendered is limited for **bus bandwidth**.
- GPU has **Video Memory,** similar to RAM for CPU, where vertices could be stored for rendering.



Send a Mesh to render: Vertex Buffer Objects

- Modern OpenGL method for sending meshes to render is through **vertex buffer objects** (**VBO**).
- Unlike regular vertex arrays, VBO haven't the restriction of being stored in RAM and sent through the bus each frame.
- With VBO, vertices can be sent to **Video Memory** once at load time and render from there each frame.



Vertex Buffer Objects

- VBO use a generic OpenGL API for buffers (note that there other types of buffers using the same functions).
- For creating and initializing a buffer:
  - glGenBuffers(size, vbos) : Creates one or more vbo
  - glBindBuffer(vbo) : Indicates what's current active buffer. Next actions will be applied to this buffer
  - glBufferData(target, size, data, usage): Initializes vbo with data
- When we are done with vertex buffer we must release it using glDeleteBuffers

Vertex Buffer Objects

- glBufferData **Usage** parameter indicates driver how are we going to use the buffer so the drive can know where to store/manage the data.
- For drawing meshes Usage parameter can be:
  - **GL_STREAM_DRAW** data is stored once and used at most a few times
  - **GL_STATIC_DRAW** data is stored once and used many times
  - **GL_DYNAMIC_DRAW** data will be modified repeatedly and used many times

Vertex Buffer Objects

VBO creation/destruction example

```
// This function must be called one time at creation of vertex buffer
unsigned CreateTriangleVBO()
{
    float vtx_data[] = { -1.0f, -1.0f, 0.0f, 1.0f, -1.0f, 0.0f, 0.0f, 1.0f, 0.0f };

    unsigned vbo;

    glGenBuffers(1, &vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);  // set vbo active
    glBufferData(GL_ARRAY_BUFFER, sizeof(vtx_data), vtx_data, GL_STATIC_DRAW);

    return vbo;
}

// This function must be called one time at destruction of vertex buffer
void DestroyVBO(unsigned vbo)
{
    glDeleteBuffers(1, &vbo);
}
```

Vertex Buffer Objects

- For rendering VBO we must specify which vertex attributes (position, normal, color, etc...) contains our buffer.
- Each Vertex Attribute is specified using the following functions
  - glEnableVertexAttribArray(index) enables a vertex attrib array slot by index. This index is a number from 0 to maximum number of attributes (depends on your GPU).
  - glVertexAttribPointer(index, size, type, normlized, stride, pointer): note that pointer is a void* for compatibility with regular vertex arrays. Pointer actually means an offset from start of buffer to find first element of the attribute.

Vertex Buffer Objects

- Finally, we do a **draw call** glDrawArrays(mode, first, count) to send buffer to render.

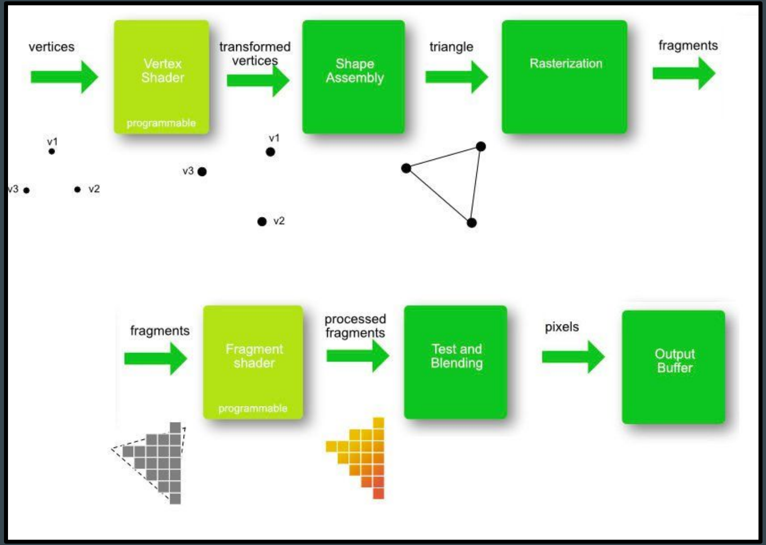
```
// This function must be called each frame for drawing the triangle
void RenderVBO(unsigned vbo)
{
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glEnableVertexAttribArray(0);

    // size = 3 float per vertex
    // stride = 0 is equivalent to stride = sizeof(float)*3
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);

    // 1 triangle to draw = 3 vertices
    glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

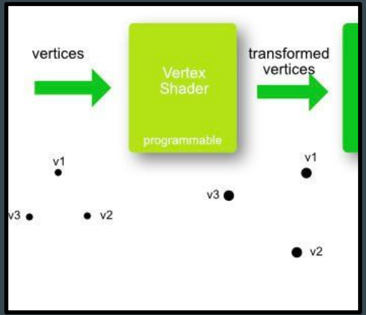
Render pipeline

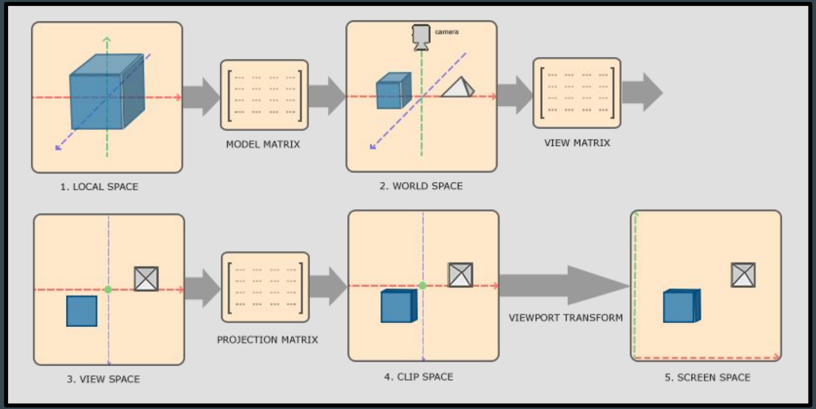
- Once our mesh is sent to GPU and just after **draw call** is done, each triangle of our mesh is processed through the **GPU render pipeline** to generate final **screen pixel colors**.



Render pipeline: vertex shader

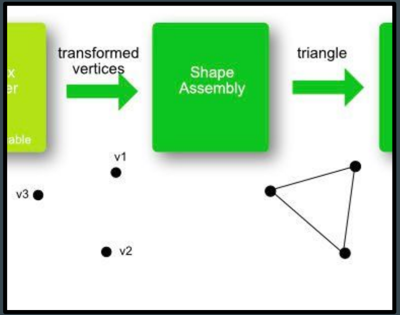
- The first stage of the GPU render pipeline, **the vertex shader,** is a **programmable stage**
- The vertex shader is responsible to transform vertex positions to **clipping space**
- Usually transform to clipping space meanus using **model view and projection** transformations.

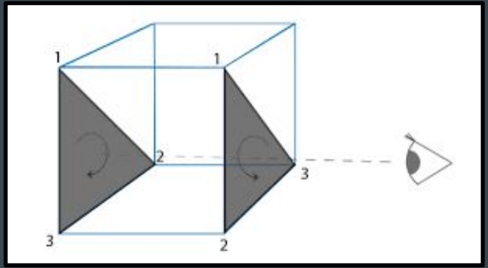




Render pipeline: shape assembly

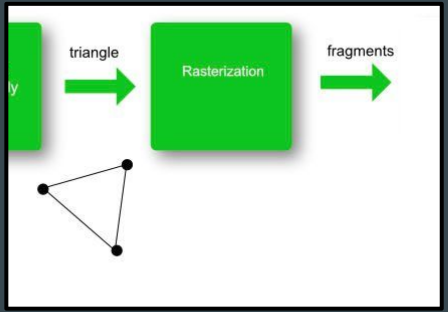
- **Shape assembly:** assemble the vertices into **primitives** (usually triangles).
- Primitives are **clipped**.
- Primitives are face culled (triangles that don't face viewer are discarded). A face is determined to be front or back by the **winding order** of the triangle vertices when it is projected in 2D.
- Finally the triangle is **screen space projected**.

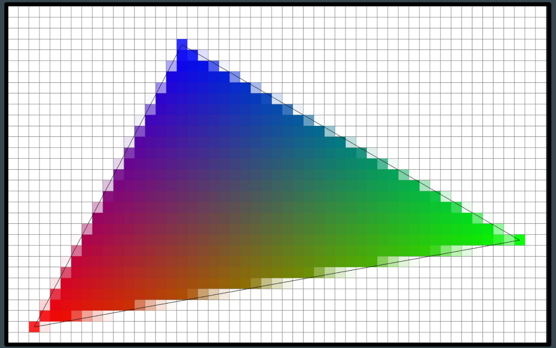




Render pipeline: rasterization

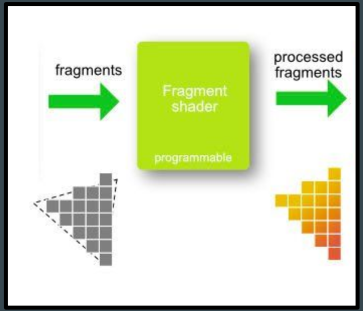
- Triangles are **discretized** into fragments.
- Fragments contains the **screen position** and the **data** needed to compute final pixel data:
  - **Interpolation of arbitrary data** coming from each vertex shader, for example the color of each vertex.
  - **Interpolation of the depth value** so we can have one per fragment.





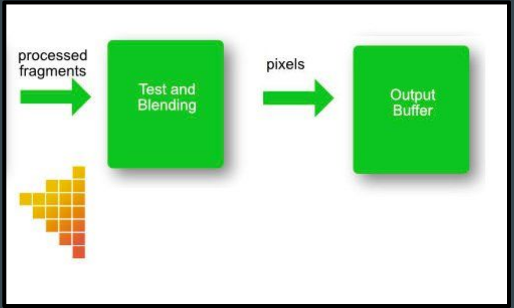
Render pipeline: fragment shader

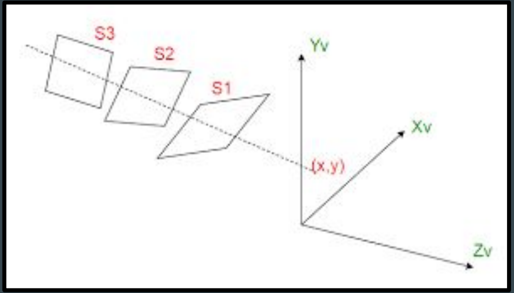
- **Fragment shader** is a **programmable stage** that receives fragments resulting from rasterization and is responsible of setting the **final color for each fragment**.
- They can also set **depth value**.
- Usually **lighting** calculations are done here.



Render pipeline: test and blending

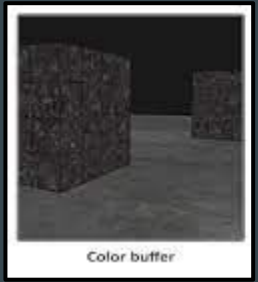
- At this stage three tests are passed: **depth test, stencil test and scissor test.**
- Depth values coming from fragments are written into **depth buffer** with the size of the viewport (screen).
- New depth values are **tested** with previously written values to check if fragments are **in front** of previously written fragments and must be finally displayed.

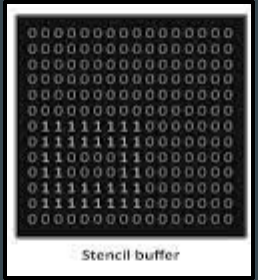


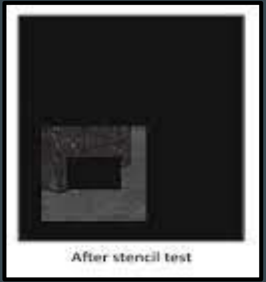


Render pipeline: test and blending

- **Stencil buffer** is an extra per pixel buffer that contains **integer data**.
- A test can be done depending on the value
- In the simplest case stencil buffer is used to **limit the area of rendering**
- It can be used **connected to depth buffer** so that stencil buffer increases/decreases its value depending on depth test fail or success.
- Stencil can be used for some effects like **silhouettes, shadow volumes, dissolves**.

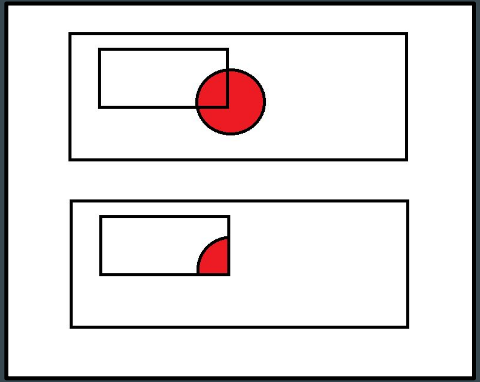


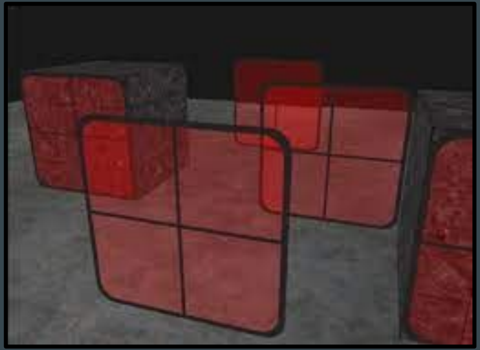




Render pipeline: test and blending

- **Scissor test** allows us to define rectangles regions on the screen so that any fragment outside of these regions will be discarded
- Final stage is **blending** where final pixel color is combined with previously written pixel.
- Blending is used to create **transparency** effects





Render pipeline: Early fragment test

- Most modern GPUs does depth and stencil test before fragment processing as an **optimization**.
- Usually fragment shader is one of the most **expensive stages** of the pipeline and being able to avoid this stage if the depth or stencil test fails can save a lot of time.
- GPUs can **disable this optimization** due to some circumstances.
- Fragment shaders can force early fragment test

Render pipeline options

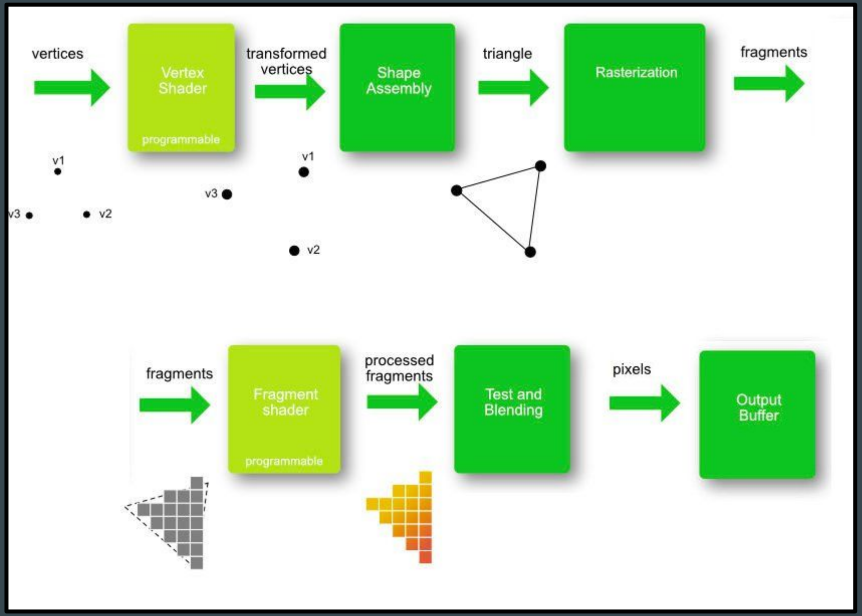
- OpenGL context contains the state of the different **options** that controls the render pipeline.
- Some of these options can be activated/deactivated using glEnable/glDisable function :
  - **GL_CULL_FACE** : For doing back face culling.
  - **GL_DEPTH_TEST :** We want enable depth test
  - **GL_SCISSOR_TEST**: We are not going to use scissor test for now.
  - **GL_STENCIL_TEST**: We are not going to use stencil test for now.

Render pipeline options

- glCullFace and glFrontFace also controls whether the cull face is **back or front face** and also what is considered a front face a **clockwise order or a counterclockwise order**. We will keep default values, culling back faces and counterclockwise order is considered front face.
- There are **more options** to activated/deactivated but we don’t need to use them for now.
- There are more functions for controlling render pipeline behaviour like for example glPolygonMode that allows you to draw meshes as wireframe.

Shaders

- We have our mesh efficiently stored in a VBO ready for being rendered
- But, remember our pipeline: 2 stages are programmable, so we need to provide a program for them:
  - **Vertex Shader**
  - **Fragment Shader**



Shaders

- Shaders are pieces of code written (OpenGL) in **GLSL language.**
- GLSL is a language quite similar to C
- Vertex and fragment shader, each one, has a **main function** → entry point
- Shaders API is focused on vector and matrix manipulation
- *"Hello world"* **vertex shader**:
    - Receives input vertex position
    - "Transforms" vertex to clippling space
- *"Hello world"* **fragment shader**:
    - Provides fragment/pixel color

Hello World Vertex Shader

- Vertex shader main function is called one time for each vertex. Each vertex attribute defined before rendering VBO is passed to the shader as a global value:

```
glEnableVertexAttribArray(0);
glVertexAttribPointer(0,3, GL_FLOAT,
GL_FALSE, 0, (void*)0);
```

Note:

1. location=n must coincide with glEnableVertexAttribArray(n) slot and with glVertexAttribPointer(n, …)
2. glVertexAttribPointer(n, 3, GL_FLOAT, …) size=3 and type=GL_FLOAT means vec3 type vertex attribute.
3. Never mind attribute name ‘my_vertex_position’

```
#version 330
layout(location=0) in vec3 my_vertex_position;

void main()
{
    gl_Position = vec4(my_vertex_position, 1.0);
}
```

A vertex shaders always has to write special global variable **gl_Position**. This variable must contain vertex position after model and view and projection transforms (clipping space). **gl_Position** is used for clipping

Hello World Fragment Shader

- Fragment shader main function is called for each fragment. Rasterization stage creates fragments (screen ‘pixels’ that are inside each polygon/triangle rendered).

One user defined **out vec4** global variable must contain color of fragment generated by the shader. A fragment shader always must output the final color. Never mind the variable name.

```
#version 330
out vec4 color;

void main()
{
    color = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Loading shaders

- Shaders are used for drawing through **programs**
- A program is a combination of, at least, two shaders: vertex, fragment.
- The process of creating a program is quite similar of a compiler creating and exe from code:
    a. Load **source code**: default_vertex.glsl and default_fragment.glsl files.
    b. **Compile** both into a shader objet
    c. **Link** both into program.

Loading shader

- We can use any standard method for loading text files: C/C++ FILE api.

```
char* LoadShaderSource(const char* shader_file_name)
{
    char* data = nullptr;
    FILE* file = nullptr;
    fopen_s(&file, shader_file_name, “rb”);

    if(file)
    {
        fseek(file, 0, SEEK_END);
        int size = ftell(file);
        data = (char*)malloc(size+1);
        fseek(file, 0, SEEK_SET);
        fread(data, 1, size, file);
        data[size] = 0;

        fclose(file);
    }

    return data;
}
```

Compile Shader

- For compiling both shaders we need to create two shader objects
  - glCreateShader(GL_VERTEX_SHADER/GL_FRAGMENT_SHADER)
  - glDeleteShader
- Once created, we can attach data and compile each one:
  - glShaderSource ← attach entire char* readed from file
  - glCompileShader ← compile
- And, of course, we can check if compilation went ok and errors:
  - glGetShaderiv ←used with GL_COMPILE_STATUS returns compilation ok.
  - glGetShaderInfoLog ← returns compilation output string useful if errors.

Compile Shader

Create and compile shader example:

```
unsigned CompileShader(unsigned type, const char* source)
{
    unsigned shader_id = glCreateShader(type);
    glShaderSource(shader_id, 1, &source, 0);
    glCompileShader(shader_id);

    int res = GL_FALSE;
    glGetShaderiv(shader_id, GL_COMPILE_STATUS, &res);
    if(res == GL_FALSE)
    {
         int len = 0;
        glGetShaderiv(id, GL_INFO_LOG_LENGTH, &len);
        if(len > 0)
        {
            int written = 0;
            char* info  = (char*)malloc(len);
            glGetShaderInfoLog(id, len, &written, info);
            LOG("Log Info: %s", info);
            free(info);
        }
    }
    return shader_id;
}
```

Program

- Finally we must create program object containing both shaders
  - glCreateProgram
  - glDeleteProgram
- Now we can attach both shaders (vertex, fragment) and link them using:
  - glAttachShader
  - glLinkProgram
- Similar to shader compilation we can check linking errors:
  - glGetProgramiv
  - glGetProgramInfoLog
- Note!!!: after linking we can delete shaders.

Program

- Create and Link Program example:

```
unsigned CreateProgram(unsigned vtx_shader, unsigned frg_shader)
{
    unsigned program_id =  glCreateProgram();
    glAttachShader(program_id, vtx_shader);
    glAttachShader(program_id, frg_shader);
    glLinkProgram(program_id);
    int res;
    glGetProgramiv(program_id, GL_LINK_STATUS, &res);
    if(res == GL_FALSE)
    {
        int len = 0;
        glGetProgramiv(program_id, GL_INFO_LOG_LENGTH, &len);
        if(len > 0)
        {
            int written = 0;
            char* info  = (char*)malloc(len);
            glGetProgramInfoLog(program, len, &written, info);
            LOG("Program Log Info: %s", info);
            free(info);
        }
    }
    glDeleteShader(vtx_shader);
    glDeleteShader(frg_shader);
    return program_id;
}
```

Program

- Now we can modify or Render VBO method to specify program to use glUseProgram

```
// This function must be called each frame for drawing the triangle
void RenderVBO(unsigned vbo, unsigned program)
{
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glEnableVertexAttribArray(0);

    // size = 3 float per vertex
    // stride = 0 is equivalent to stride = sizeof(float)*3
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);

    glUseProgram(program);
    // 1 triangle to draw = 3 vertices
    glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

Exercise

- Initialize render pipeline options at **ModuleRender::Init** method
- Create a **ModuleProgram** in our Engine project able to create a program form to files (vertex and fragment shader files)
- Create a **ModuleRenderExercise** that
    - At Init method, loads a triangle into a VBO with vertices: (-1, -1, 0) (1, -1, 0) (0, 1, 0)
    - At Init method creates a program with Hello World vertex and fragment shaders
    - At Update method renders VBO triangle using Hello World program.
- Good Tutorial and good starting point

Debugging OpenGL

- Programming OpenGL is hard because any error passing arguments to functions or a wrong order of function calls may cause an unexpected result on render.
- We need a way to get errors in our OpenGL function calls
- glGetError returns an error if something wrong happened in previous function call.
- But calling glGetError after each function call is tedious

Debugging OpenGL

- There is a method to set a callback function for being notified on any error but we need to configure OpenGL context properly:
  - Needs enable context debug flags:
    - **SDL_GL_SetAttribute**(SDL_GL_CONTEXT_FLAGS, SDL_GL_CONTEXT_DEBUG_FLAG);

Debugging OpenGL

- At OpenGL **initialization** we need:
  - glEnable(GL_DEBUG_OUTPUT); → Enable Output Callbacks
  - glEnable(GL_DEBUG_OUTPUT_SYNCHRONOUS); → Output Callbacks
  - glDebugMessageCallback(&OurOpenGLErrorFunction, nullptr); → sets the callback
  - glDebugMessageControl(GL_DONT_CARE, GL_DONT_CARE, GL_DONT_CARE, 0, nullptr, true); → filters notifications
- Note: Output Callbacks are expensive so, they should be initialized only on Debug versions

Debugging OpenGL

- Implement OurOpenGLErrorFunction. Example:

```
void __stdcall OurOpenGLErrorFunction(GLenum source, GLenum type, GLuint id, GLenum severity, GLsizei length, const GLchar* message, const void* userParam)
{
    const char* tmp_source = "", *tmp_type = “”, *tmp_severity = “”;
    switch (source) {
        case GL_DEBUG_SOURCE_API:              tmp_source = "API"; break;
        case GL_DEBUG_SOURCE_WINDOW_SYSTEM:   tmp_source = "Window System"; break;
        case GL_DEBUG_SOURCE_SHADER_COMPILER: tmp_source = "Shader Compiler"; break;
        case GL_DEBUG_SOURCE_THIRD_PARTY:     tmp_source = "Third Party"; break;
        case GL_DEBUG_SOURCE_APPLICATION:     tmp_source = "Application"; break;
        case GL_DEBUG_SOURCE_OTHER:           tmp_source = "Other"; break;
    };
    switch (type) {
        case GL_DEBUG_TYPE_ERROR:               tmp_type = "Error"; break;
        case GL_DEBUG_TYPE_DEPRECATED_BEHAVIOR: tmp_type = "Deprecated Behaviour"; break;
        case GL_DEBUG_TYPE_UNDEFINED_BEHAVIOR:  tmp_type = "Undefined Behaviour"; break;
        case GL_DEBUG_TYPE_PORTABILITY:         tmp_type = "Portability"; break;
        case GL_DEBUG_TYPE_PERFORMANCE:         tmp_type = "Performance"; break;
        case GL_DEBUG_TYPE_MARKER:              tmp_type = "Marker"; break;
        case GL_DEBUG_TYPE_PUSH_GROUP:          tmp_type = "Push Group"; break;
        case GL_DEBUG_TYPE_POP_GROUP:           tmp_type = "Pop Group"; break;
        case GL_DEBUG_TYPE_OTHER:               tmp_type = "Other"; break;
    };
    switch (severity) {
        case GL_DEBUG_SEVERITY_HIGH:         tmp_severity = "high"; break;
        case GL_DEBUG_SEVERITY_MEDIUM:       tmp_severity = "medium"; break;
        case GL_DEBUG_SEVERITY_LOW:          tmp_severity = "low"; break;
        case GL_DEBUG_SEVERITY_NOTIFICATION: tmp_severity = "notification"; break;
    };
    LOG("<Source:%s> <Type:%s> <Severity:%s> <ID:%d> <Message:%s>\n", tmp_source, tmp_type, tmp_severity, id, message);
}
```

Debugging OpenGL documentation

- Tutorial on using glDebugMessageCallback
- Another useful link
- Why is my window black ? link to useful tests if you don't see expected rendering results.