6

Building responsive UI interface

In this chapter, we will learn and build the basics of constructing a high performance user interface from scratch. A User interface plays a vital role in any application, it’s a medium through which a user not only interacts with the application but also observe various states of application via visualizations. There is an obvious question that comes to mind – what is need to building UI interface from Vulkan or any other low level API? When it comes to rendering complex 1000’s UI objects with real-time rendering visualization and interaction at the same time, the traditional API might not sufficient enough for responsive rendering rates. Also, each ui object might bring a certain amount of overhead which might not a required for the custom applications. Therefore, the learning in this chapter is not to encourage replacement of any existing GUI libraries but to overcome situation where traditional GUI encounter limitations. The other reason is to build custom UI that provides an application a dedicated fast path without indulging into external overheads with better utilization of GPU.

Building Ui interface is vast and complex topic, this chapter introduce the basic concepts and demonstrate building simple and custom UI’s without dropping the performance. We will start with geometric instancing and learn to draw multiple objects in a single command. Next, we will build relation between different rendering object with Scene-Graph, it’s a logical concept that manages a 2D/3D scene and manages transformation graph. We will manage application event’s such as mouse event and handle interaction with objects.

In this chapter, we will learn step-by-step how to usecover the following topics and by the end of the chapter you should be able to run your first Vulkan application to render a triangle on your system.

* Geometric Instancing – Rendering multiple object in single command
* Getting started with Scene-Graphs
* Transformation Graph - preserving Parent-Child transformation.
* Create complex models with transformation graph
* Implementing a scene with multiple views
* Event management Mouse interaction – hover, cl
* Optimization and scaling
* Summary

# Geometric Instancing

Geometric instancing is feature rendering multiple instances of the same object under single API call. Instancing is helpful in rendering same types of objects in application like particle system, vegetation, crowd simulations etc.

This example demonstrates rendering of 1,000,000 rectangle item with the help of geometric instancing. Geometric instancing consist of two type of attributes – attributes that are constant and those changes per instance. For example, the rectangle attributes geometry attributes like vertices position and vertex color are *constant*, however the spatial position of each rectangle and orientation are *different*.

For this example, we are reusing the Ch4\_01c\_SimpleCube\_EnableDepthBuffer example from chapter 2, <chapter name>. We have modified the geometry to render the rectangle instead. Let’s look at the step-by-step process to implement geometric instancing.

1. For each instance of rectangle store the transformation information containing the unique position of the rectangle in the 3D space in user define data structure InstanceData.

**// VulkanHelper .h**

struct VulkanBuffer

{

VkBuffer m\_Buffer; **// Buffer resource object**

uint64\_t m\_DataSize; **// Actual data size request for, use**

**// m\_MemRqrmnt.size for actual**

**// backing size**

VkDeviceMemory m\_Memory; **// Buffer resource object's**

**// allocated device memory**

VkMemoryRequirements m\_MemRqrmnt; **// Memory requirement for**

**// the allocation buffer, useful in mapping/unmapping**

VkMemoryPropertyFlags m\_MemoryFlags; **// Memory property flag**

};

**// Rect.h**

**// Per-instance data block**

struct InstanceData { glm::mat4 MVP; };

VulkanBuffer m\_InstanceBuffer; **// Instance data GPU buffer**

void PrepareInstanceData();

1. During the initialization of the rectangle class the instance data is prepared for 1,000,000 rectangle objects in PrepareInstanceData(). The position information is stored in the instanceData variables and uploaded on the GPU with the help of staging buffer.

In staging, two different memory regions are used for the physical allocation. Usually, the first memory region is host-visible and the second is device-local and is the ideal memory placement for a resource which may not be visible to the host. The application must first populate the resource in a staging buffer that is host-visible, and then transfer it to the ideal location using special copying buffer api’s.

void Rect::PrepareInstanceData()

{

std::vector<InstanceData> instanceData;

instanceData.resize(INSTANCE\_COUNT);

std::mt19937 rndGenerator(time(NULL));

std::uniform\_real\_distribution<double> uniformDist(0, 1);

for (auto i = 0; i < INSTANCE\_COUNT; i++)

{

const float theta = 2 \* M\_PI \* uniformDist(rndGenerator);

const float phi = acos(1 - 2 \* uniformDist(rndGenerator));

glm::vec3 pos = glm::vec3(30 \* sin(phi) \* cos(theta),

40 \* sin(theta) , cos(phi)) \* 20.0f;

instanceData[i].MVP = glm::translate(glm::mat4(1.0f),pos);

}

VkMemoryPropertyFlags memoryProperty =

VK\_MEMORY\_PROPERTY\_DEVICE\_LOCAL\_BIT;

m\_InstanceBuffer.m\_MemoryFlags = memoryProperty;

m\_InstanceBuffer.m\_DataSize = instanceData.size() \*

sizeof(InstanceData);

VulkanHelper::CreateStagingBuffer(

m\_VulkanApplication->m\_hDevice,

m\_VulkanApplication->m\_physicalDeviceInfo.memProp,

m\_VulkanApplication->m\_hCommandPool,

m\_VulkanApplication->m\_hGraphicsQueue,

m\_InstanceBuffer,

VK\_BUFFER\_USAGE\_VERTEX\_BUFFER\_BIT |

VK\_BUFFER\_USAGE\_TRANSFER\_DST\_BIT,

instanceData.data());

}

The instance data upload is stored into device local memory buffer m\_InstanceBuffer.

1. The below italic bold changes indicated in the existing vertex shader supports geometric instancing. There is no change required for fragment shader.

The vertex shader is injected with vertex input with two type of rates:

1. Per vertex: The inPosition and inColor are per vertex attribute are bind to 0th index and at layout index 0 and 1 respectively. This vertex input data is common to all the instances.
2. Per instance: The type of rate accessed from the uploaded GPU buffer per object or instanced based. For example: instancePos, at bind index 1.

How come the binding index for inPosition and inColor is 0 and instancePos is 1?

When binding index is not explicitly indicated, in that case the shader complier automatically assigns the buffer index in the order in which they are specified. In staging, two different memory regions are used for the physical allocation. Usually, the first memory region is host-visible and the second is device-local and is the ideal memory placement for a resource which may not be visible to the host. The application must first populate the resource in a staging buffer that is host-visible, and then transfer it to the ideal location using special copying buffer api’s.

// Filename: RectInstance.vert

#version 450

#extension GL\_ARB\_separate\_shader\_objects : enable

layout (std140, binding = 0) uniform TransformBufferStruct{

mat4 mvp;

} TransformBuffer;

**// Vextex attributes**

layout (location = 0) in vec4 inPosition;

layout (location = 1) in vec4 inColor;

**// Instance attributes**

***layout (location = 2) in mat4 instancePos;***

**// Attribute to the next stage**

layout(location = 0) out vec4 fragColor;

out gl\_PerVertex {

vec4 gl\_Position;

};

void main()

{

fragColor = inColor;

***gl\_Position = TransformBuffer.mvp \* instancePos\* inPosition;***

gl\_Position.z = (gl\_Position.z + gl\_Position.w) / 2.0;

}

1. Fsdsf
2. Sf
3. S
4. Fs
5. f

for this we use 1000 matrices in a VBO, each matrix contains a transformation to place a cube in 3D space. The information of the matrices are updated using range map buffer feature as discussed in the previous recipe. This allows to pass new transformation data on-the-fly at run time, the transformed data contains new rotation and translated positions.

Insert image B06926\_01.png

## How to do it...

So far in our recipes, the Model-View-Projection matrix is always treated as uniforms in the vertex shader, for this recipe we make the use the VAO and declares the Model-View-Projection matrix as a generic attribute instead of a uniform. Since the matrix is an attribute a new VBO is required, this VBO is stored in variable matrixId, the RenderCube() uses the map buffer to update transformation matrix data.

Below are the steps describes to implement geometric instancing:-

1. Create the vertex shader and add the below code. There is no change required for fragment shader, it can be reused.

#version 300 es

layout(location = 0) in vec4 VertexPosition;

layout(location = 1) in vec4 VertexColor;

**layout(location = 2) in mat4 MODELVIEWPROJECTIONMATRIX;**

out vec4 Color;

void main() {

gl\_Position = MODELVIEWPROJECTIONMATRIX \* VertexPosition;

Color = VertexColor;

}

1. In the Cube::InitModel() use the existing code and add a new VBO for matrix transformation, get the ID of the generated buffer object in matrixId.

**// Create VBO for transformation matrix**

glGenBuffers(1, &matrixId);

glBindBuffer (GL\_ARRAY\_BUFFER, matrixId);

1. Allocate the memory to the VBO for matrix transformation. The dimension variable is initialized with 10, it gives the number of cubes along an axis. Therefore, along X, Y and Z axises 10x10X10 = 1000 cubes, the total size of the buffer would be: sizeof (GLfloat) \* 16 (16 float element in mat4) \* 1000 (cubes).

glm::mat4 transformMatrix[dimension][dimension][dimension];

glBufferData(GL\_ARRAY\_BUFFER, sizeof(transformMatrix) , 0, GL\_DYNAMIC\_DRAW);

The glBufferData uses GL\_DYNAMIC\_DRAW, this symbolic constant specifies that the buffer is going to contain some data which is dynamic in nature. In other word, the data will require updates in the buffer. This symbolic constant helps the graphics driver to manage buffer memory in the best possible way to achieve high performance graphics rendering.

1. In the same function, after creating the vertex array object (Vertex\_VAO\_Id) define the generic attribute states and configuration of the transformation matrix buffer object. This help is saving the vertex array client states and the buffer binding in the VAO (Vertex\_VAO\_Id). The glVertexAttribDivisor calculates instance ID from the given total number of instances, for more information, please refer to *There More…* section in this recipe.

**// Create VBO for transformation matrix and set attributes**

glBindBuffer( GL\_ARRAY\_BUFFER, matrixId );

glEnableVertexAttribArray(MATRIX1\_LOCATION);

glEnableVertexAttribArray(MATRIX2\_LOCATION);

glEnableVertexAttribArray(MATRIX3\_LOCATION);

glEnableVertexAttribArray(MATRIX4\_LOCATION);

glVertexAttribPointer(MATRIX1\_LOCATION,4,GL\_FLOAT,GL\_FALSE,

sizeof(glm::mat4),(void\*)(sizeof(float)\*0));

glVertexAttribPointer(MATRIX2\_LOCATION,4,GL\_FLOAT,GL\_FALSE,

sizeof(glm::mat4),(void\*)(sizeof(float)\*4));

glVertexAttribPointer(MATRIX3\_LOCATION,4,GL\_FLOAT,GL\_FALSE,

sizeof(glm::mat4), (void\*)(sizeof(float)\*8));

glVertexAttribPointer(MATRIX4\_LOCATION,4,GL\_FLOAT,GL\_FALSE,

sizeof(glm::mat4), (void\*)(sizeof(float)\*12));

glVertexAttribDivisor(MATRIX1\_LOCATION, 1);

glVertexAttribDivisor(MATRIX2\_LOCATION, 1);

glVertexAttribDivisor(MATRIX3\_LOCATION, 1);

glVertexAttribDivisor(MATRIX4\_LOCATION, 1);

1. In the Cube::RenderCube() use range buffer mapping to map the transformation buffer on the client side memory. Update the data in the memory and unmap it. Use VAO and render the cube of cubes using geometric instance API called glDrawElementsInstanced. This API’s last argument specifies the number of instances the given primitive will be rendered.

void Cube::RenderCube()

{

glBindBuffer( GL\_ARRAY\_BUFFER, matrixId );

glm::mat4\* matrixBuf = (glm::mat4\*)glMapBufferRange

(GL\_ARRAY\_BUFFER, 0, sizeof(glm::mat4\*)\*(dimension \*dimension\*dimension), GL\_MAP\_WRITE\_BIT);

static float l = 0;

TransformObj->TransformRotate(l++, 1, 1, 1);

TransformObj->TransformTranslate

(-distance\*dimension/4,-distance\*dimension/4, -distance\*dimension/4);

glm::mat4 projectionMatrix = \*TransformObj->

TransformGetProjectionMatrix();

glm::mat4 modelMatrix = \*TransformObj->

TransformGetModelMatrix();

glm::mat4 viewMatrix = \*TransformObj->

TransformGetViewMatrix();

int instance= 0;

for ( int i = 0; i < dimension; i++ ){

for ( int j = 0; j < dimension; j++ ){

for ( int k = 0; k < dimension; k++ ){

matrixBuf[instance++] = projectionMatrix \*

viewMatrix \* glm::translate(modelMatrix, glm::vec3( i\*distance , j\*distance, k\*distance)) \* glm::rotate( modelMatrix, l, glm::vec3(1.0, 0.0, 0.0));

} } }

glUnmapBuffer ( GL\_ARRAY\_BUFFER );

glBindVertexArray(Vertex\_VAO\_Id);

glDrawElementsInstanced(GL\_TRIANGLES,36,

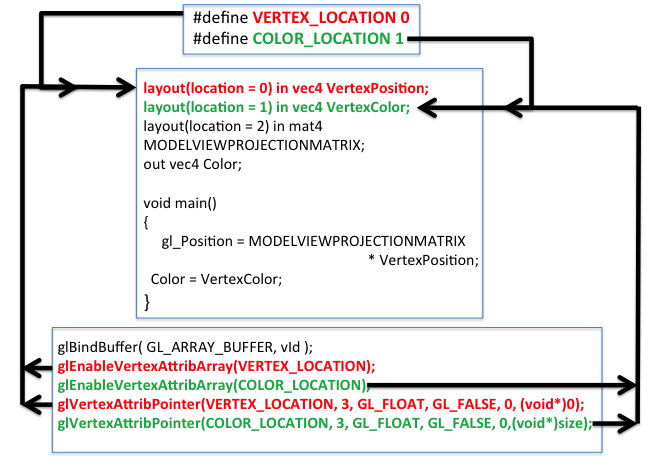
GL\_UNSIGNED\_SHORT, (void\*)0, dimension\*dimension\*dimension);

}

## How it works...

The application first compiles the shader programs, this allows to aware of all generic attribute locations used in the shader program. Create a VBO of 1000 matrix elements, each element represents a transformation matrix, this matrix element is updated with new values of the transformation every frame in the RenderCube function.

The generic attributes are first enabled using glEnableVertexAttribArray, the data array is attached to the generic location with glVertexAttribPointer. The below image shows how the OpenGL ES program API is attached to the layout location of the vertex shader to send data.

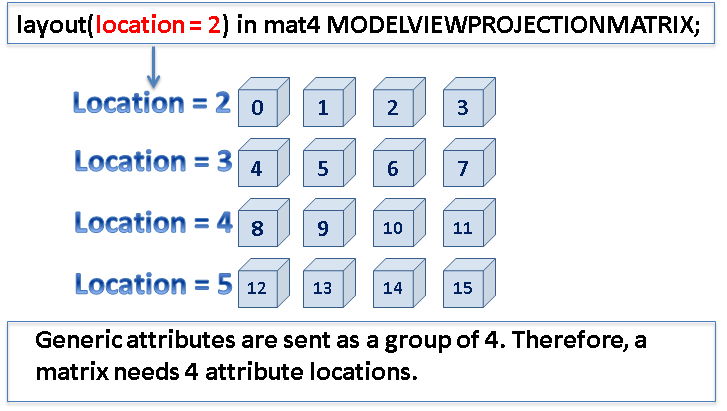


Insert image 5527OT\_03\_02.png

This should be noted that the generic attributes are sent as a group of 4, therefore for a 4x4 matrix we need 4 attribute locations. The start location of the attribute should be mention into the vertex shader using layout qualifier.

layout(location = 2) in mat4 MODELVIEWPROJECTIONMATRIX;

The below image shows how the attribute locations are managed by the compiler.



Inser6t image 5527OT\_03\_03.png

Similar to the other locations- VERTEX\_LOCATION(0) andCOLOR\_LOCATION (1), the transformation matrix locations (2,3,4,5) are also needed to be enabled and attached to the array data.

The glVertexAttribDivisor API is responsible for controlling the rate at which OpenGL ES advances the data from an instanced array. The first parameter of this API specifies the generic attribute that needs to be treated as an instanced array. This tells the OpenGL ES pipeline to use this attribute per instance rendering. For example, in this example the generic attributes 2, 3, 4, 5 are instanced attributes. Therefore, OpenGL ES consumes the data from the transformation matrix array of matrix as an instance ID. We will see in a moment how this instance ID is calculated.

The default value of the divisor is 0 when it is not specified in the program explicitly. If the divisor is zero, the attribute index is advanced once per vertex. If the divisor is non-zero, the attribute advance once per divisor instance of the set(s) of the vertices being rendered.

**Syntax**

void glVertexAttribDivisor(GLuint index, GLuint divisor);

|  |  |
| --- | --- |
| Index | This specifies generic attribute layout location |
| Divisor | Specify the number of instances that will pass between updates of the generic attribute at slot index. |

The rendering of the geometric instancing requires special instanced based drawing APIs from OpenGL ES 3.0 as mentioned below for array and index based geometric data.

**Syntax:**

void glDrawElementsInstanced(GLenum mode, GLsizei count,

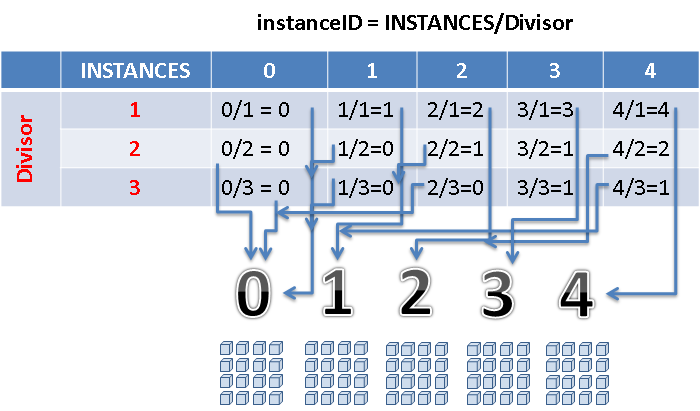
GLenum type, const void \* indices, GLsizei primcount);

|  |  |
| --- | --- |
| mode | This specifies type of the primitive that needs to be rendered |
| count | Specifies the number of indices considered in the drawing |
| type | Used by glDrawElementsInstanced, specify the data type of the indices stored |
| indices | Specify the arrays containing order of the indices |
| primCount | Specifies the number of copies to be rendered |

In the present recipe, the glDrawElementsInstanced API is used to render multiple instances of the same object, this API works in conjunction with another API called glVertexAttribDivisor. In order to update the VBO matrix elements, buffer mapping is used which is an efficient way to update the buffer elements. If the geometric data not index but it is array based then glDrawArraysInstaced can be used, which accepts almost same parameters, please refer to online OpenGL ES 3.0 reference manual for more info.

## There’s more...

The second attribute of glVertexAttribDivisor specifies the divisor, this divisor helps in calculating instance ID from the total number of instances. The below image shows a trivial example of working logic of this API, in this we have assumed that there are a total 5 instances to be rendered and it contains 5 matrices. When Divisor is 5 – it produced 5 instance ID of the (0, 1, 2, 3, 4) this instance ID will be used as an index to Transformation Matrix array. Similarly, for the case of 2 it generates 3 instances (0, 1, 2) and 2 instances(0,1) when it is 3.



Insert image 5527OT\_03\_04.png