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 in which multiple instances of the same object drawn under a 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.

In order to implement Geometric instancing one need to understand what properties of the application is dynamically changing and what remain constant. Therefore, we can categories an application properties into type types:-

1. Constant properties: These properties do not change frequently, therefore it can stored in the fast access device memory in the form of Uniform buffers. For example, in this application our rectangle’s spatial position and color information are *constant.*
2. Dynamic properties: These type of properties changes very frequently therefore the CPU constantly bugs the GPU to update its memory contents. As a result, these properties are store in a GPU memory provide a faster execution path to update the GPU memory. Since this memory is visible to the host it may add some latency for GPU to read the data from the invalidate memory region after an update.

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 of Cube and Similar the class name is rename to Rect. Let’s look at the step-by-step process to implement geometric instancing.

1. For each instance of rectangle, the transformation information is stored in InstanceData data structure. This contains the unique position of the rectangle in the 3D space in the form of translate information. During the initialization, the location of each rectangle is computed and store in a CPU contiguous memory from where it is uploaded into the GPU memory using m\_InstanceBuffer .

**// 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; };

**// Instance data GPU buffer**

VulkanBuffer m\_InstanceBuffer;

**// 0th for vertex buffer binding, 1st for instancing buffer**

VkVertexInputBindingDescription m\_VertexInputBinding[2];

**// Why 6? = 2(for position and color) + 4 (transform matrix)**

VkVertexInputAttributeDescription m\_VertexInputAttribute[6];

void PrepareInstanceData();

1. **Binding resources Interpreting vertex input data:** The host supplies two type of resources which are consumed by the device at two different rates *per vertex* and *per instance*.

These resource are interpreted using vertex input binding index, the below code shows two vertex input binding variable are used for hooking per vertex and per instance information.

**// Indicates the rate at which the information will be**

**// injected for vertex input.**

m\_VertexInputBinding[0].binding = VERTEX\_BUFFER\_BIND\_IDX;

m\_VertexInputBinding[0].inputRate =

VK\_VERTEX\_INPUT\_RATE\_VERTEX;

m\_VertexInputBinding[0].stride = dataStride;

m\_VertexInputBinding[1].binding = INSTANCE\_BUFFER\_BIND\_IDX;

m\_VertexInputBinding[1].inputRate=

VK\_VERTEX\_INPUT\_RATE\_INSTANCE;

m\_VertexInputBinding[1].stride = sizeof(InstanceData);

1. **Interpreting vertex input attributes:** For these two type of resources, the GPU need to understand the format in order to interpret the incoming attributes correctly. The per vertex attributes – position and color are both are 96-bit signed floating-point format that has a 32-bit R component, a 32-bit G component and a 32-bit B component and hooked to location 0 and 1 respectively at binding at 0 (VERTEX\_BUFFER\_BIND\_IDX).

**// The VkVertexInputAttribute interpreting the vertex data.**

m\_VertexInputAttribute[0].binding = VERTEX\_BUFFER\_BIND\_IDX;

m\_VertexInputAttribute[0].location = 0;

m\_VertexInputAttribute[0].format = VK\_FORMAT\_R32G32B32\_SFLOAT;

m\_VertexInputAttribute[0].offset=offsetof(Vertex,m\_Position);

m\_VertexInputAttribute[1].binding = VERTEX\_BUFFER\_BIND\_IDX;

m\_VertexInputAttribute[1].location = 1;

m\_VertexInputAttribute[1].format = VK\_FORMAT\_R32G32B32\_SFLOAT;

m\_VertexInputAttribute[1].offset=offsetof(Vertex, m\_Color);

1. **Interpreting vertex input data:** The transformation matrix is per instance where each row is stored at separate location as shown below:

**/ The VkVertexInputAttribute interpreting instancing data.**

m\_VertexInputAttribute[2].binding = INSTANCE\_BUFFER\_BIND\_IDX;

**m\_VertexInputAttribute[2].location = 2;**

m\_VertexInputAttribute[2].format=VK\_FORMAT\_R32G32B32A32\_SFLOAT

m\_VertexInputAttribute[2].offset = 0;

m\_VertexInputAttribute[3].binding = INSTANCE\_BUFFER\_BIND\_IDX;

**m\_VertexInputAttribute[3].location = 3;**

m\_VertexInputAttribute[3].format=VK\_FORMAT\_R32G32B32A32\_SFLOAT

m\_VertexInputAttribute[3].offset = 16 \* 1;

m\_VertexInputAttribute[4].binding = INSTANCE\_BUFFER\_BIND\_IDX;

**m\_VertexInputAttribute[4].location = 4;**

m\_VertexInputAttribute[4].format=VK\_FORMAT\_R32G32B32A32\_SFLOAT

m\_VertexInputAttribute[4].offset = 16 \* 2;

m\_VertexInputAttribute[5].binding = INSTANCE\_BUFFER\_BIND\_IDX;

**m\_VertexInputAttribute[5].location = 5;**

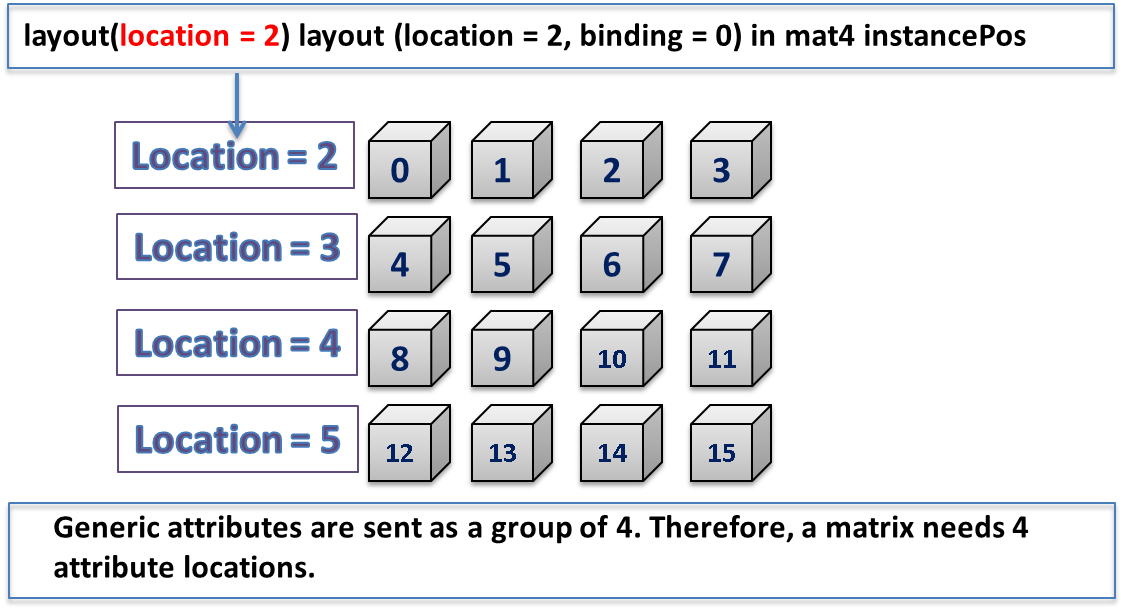
m\_VertexInputAttribute[5].format=VK\_FORMAT\_R32G32B32A32\_SFLOAT

m\_VertexInputAttribute[5].offset = 16 \* 3;

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, binding = 0) in mat4 instancePos;

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



1. The vertex shader is injected with vertex input with two type of rates:
2. **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.
3. **Per instance:** The type of rate accessed from the uploaded GPU buffer per object or instanced based. For example: instancePos, at bind index 1.

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

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. 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,

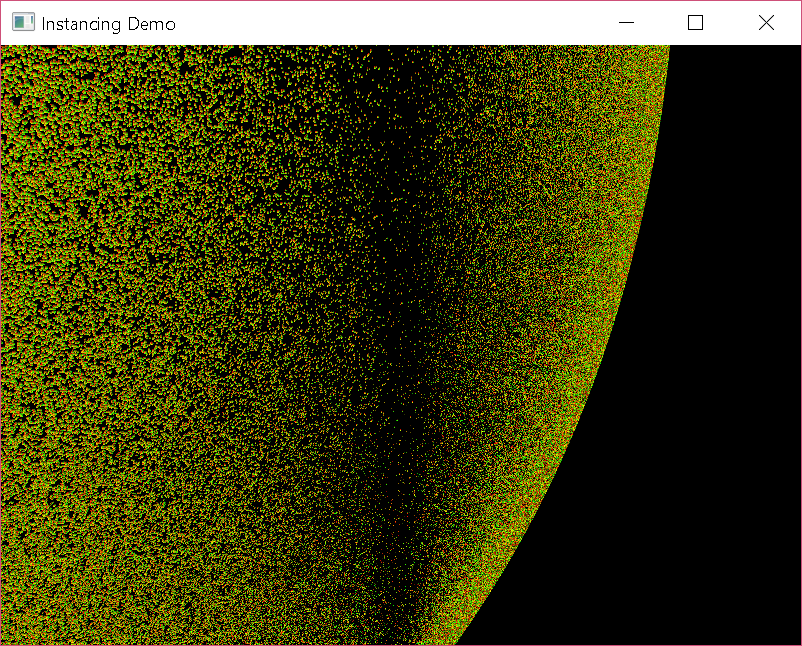
VK\_BUFFER\_USAGE\_VERTEX\_BUFFER\_BIT |

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

instanceData.data());

}

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



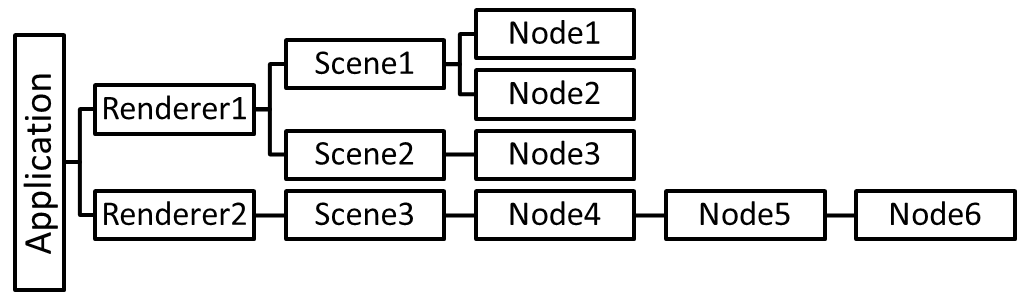
# Scene Graphs

Scene-Graph paradigm is concept for managing large complex scene efficiently. The scene graph encapsulates the hierarchical structure of a complete 3D scene, it mainly has two aspects- semantics and rendering. The semantic aspect works like a database which manages the visual representation and state management, think of this like a visual database which tells the graphical system which scene is going to come and which scene is not being under use so that it can be released along with its resources for better optimization and memory management. On the other hand, the rendering aspect deals with the life cycle management of drawable entities or models which includes the initialization, deinitialization, processing, control management and displaying them on the screens.

The scene graph is a big and evolving topic, covering all the aspects of requirements is out of the scope of this title. Mainly in this chapter, we will create a small architecture that allows to manage multiple scenes, each scene can consist of multiple lights, cameras, models. Complex models can be created using the parent-child relationship with the help of local and relative transformations. Models can be applied to predefined materials dynamically and all this will be done outside the graphics engine in a separate C++ file, this will keep the scene-graph hierarchy logic preserve at single place to manage it easily.

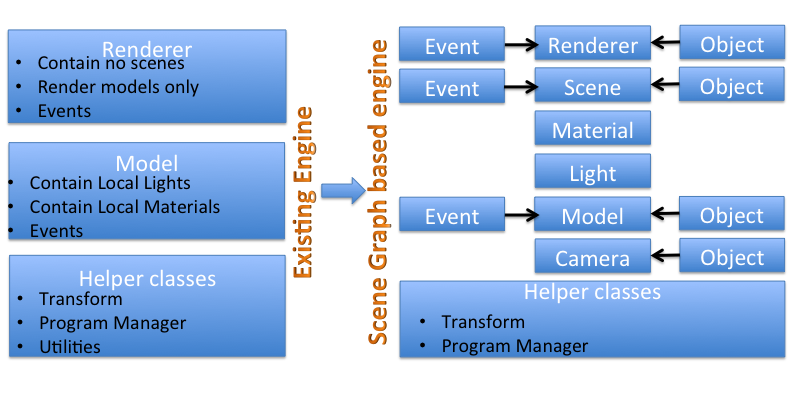
For real time 3D applications, we need to extend the design to meet the scene graph architecture requirements:-

1. **Hierarchical relationship:** Various modules of the system can be arranged in a hierarchical fashion. The *Application* contains the *Renderer* which comprises of various scenes containing one or millions of nodes. A node represent a render able item.



1. **Objects with parent-child relationship**: The nodes must support the parent-child relationship. In the parent-child relationship, a parent manages all its children automatically, this way the semantics and rendering can be managed in an optimized way.
2. **Transform Graphs**: Each nodes in the system stored the transformation with respect to its parent.
3. **Multiple scene management:** The system must allow multiple scenes to render. Each scene, may have its own projection system to display the nodes under different viewpoints. This is also a good place to store common resources like camera, lights or transformation that can be applied to all contained nodes.
4. **Separating semantic and rendering:** The rendering of the objects must be loosely coupled with the semantics. The rendering output could be affected by a number of factors like change in state, user input or both. The design should be flexible enough for managing the states and events.
5. **Level Of Detail (LOD):** The LOD uses the computed information of an object about how far is from the camera view or an observer. If the object lies outside the viewing frustum then it can be ignored before it consumes vital resources of the system. The object which is in frustum view, but far-away from the camera can be rendered at lower fidelity where fewer polygons and smaller textures can be used.
6. **State Enacapsulation:** It’s important that each node or object in the system contains the state that could be able to reveal the nature of the object. This way several similar types of objects can be clubbed together by traversing the parent-child hierarchy, this will be highly efficient in avoiding random state switches. For example, texture loading and binding.

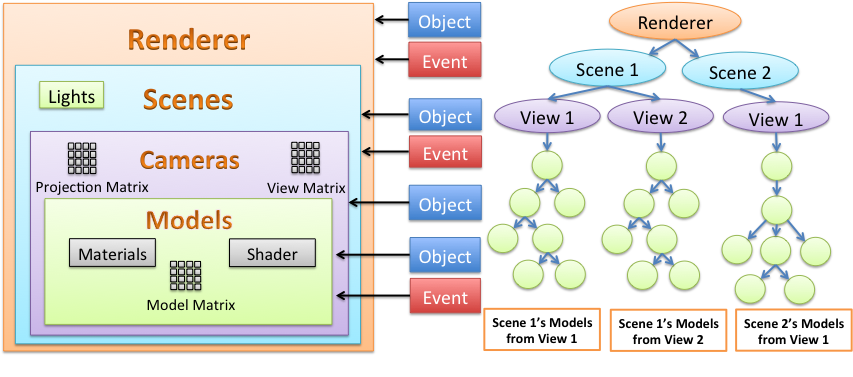
The below image shows the block diagram of the existing engine (left) and the expected scene graph paradigm (right). The scene graph design is segregated in many simpler reusable modules, where each module is self explanatory in the image itself. The block represented by the *Object* module is a base class for most of the other modules those exhibits the parent-child relationship, similarly modules which support the event handling must be inherited from the *Event*.



5527\_10\_1.png

In another image below you can see the hierarchical relationship between the different modules in the scene graph. The Renderer is the graphics engine that contains various scenes, these scenes can be added and removed to the rendering engine dynamically. A scene contains one or more cameras as per requirement, it also contains the models that the scene needs to render. These models are rendered from different camera views which can be used to create a multiple view scene or a single scene but viewing from different positions.

The transformation is managed in Model-View-Projection analogy where the modeling transformation is carried out in the Models, the projection and viewing transformation is calculated in the Camera’s module. Each Model exhibits a parent-child relationship where the parent is fully responsible for managing the life cycle of their children. The events in the system flow in the top-down fashion, the native application receives the events and pass it to the Renderer which further propagates the event to the scene. The Scene detects the view to which the event belongs to and the events is sent to all corresponding models in the view where it is finally handled.



5527\_10\_2.png

This chapter will take us in a systematic approach to develop the scene graphs:-

1. **Implementing the first scene in the scene graph (Recipe 1):** This recipe builds the foundation of scene-graph in which it supports Scene, Model, Light and Material module.The modeling will be done outside the rendering engine in the NativeTemplate.cpp.
2. **Adding local and relative transformation (Recipe 2):** This recipe will introduce the local and the relative transformation concept to the existing scene graph. The local transformation is only applicable within the renderable object, however,the relative transformation is received from a parent and propagated to children.
3. **Adding Parent-Child support in the scene graph (Recipe 3):** This recipe builds the parent-child relationship between similar types of objects.
4. **Create complex models with transformation graph (Recipe 4):** This recipe will make use of the previous recipe concepts and demonstrate how to build complex animated models like a revolving windmill.
5. **Implementing picking with ray trace technique (Recipe 5):** This recipe will add the support of events to the scene graph and help in implementing the ray trace based picking technique that’s allows to select 3D objects in a scene.
6. **Implementing 2D textured button widgets (Recipe 6):** Implementing 2D widgets using screen coordinate system, this recipe contains another sub-recipe which implements the clicking of the button widget.
7. **Navigating the scene with camera system (Recipe 7):** This recipe will implement the camera support to the scene.
8. **Implementing a scene with multiple views (Recipe 8):** This recipe enables the scene graphics to render multiple views of a single scene.