**A1: Scene Viewer**

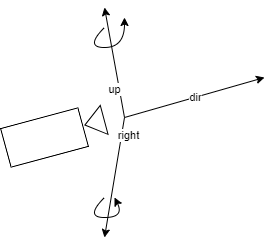
by Shengjian Chen (Michael)

This Scene Graph Viewer assignment can be divided into 4 parts: 1) argument parsing and file reader; 2) Main logic of Vulkan Setup and Graphic Pipeline; 3) back library support (cglm, mcjp); 4) Managing data for each frame. The code was structured based on what its affect on. i.e. for those creating vertex buffer and passing vertex buffer’s data, I put them all together into one vertex\_buffer.cpp file. This structure helps me quickly target where I should implement and update in around 5000 loc. Besides, I have actually 2 types of implementation Instance Drawing(using uniform buffer to pass model matrix) and Vertex Drawing(copy data to vertex buffer each frame). These are in two different branches, Instance Drawing – uniform\_buffer\_branch and Vertex Drawing – main\_branch.

My Animation:

*Control the Scene Viewer:*

Most of the command is through keyboard input. I introduce camera control first, each camera would have two corresponding vector “dir” and “up”, we can use cross-multiply to have its “right”. Only camera named “debug” and “usr” can be controlled, (in my implementation, better use “debug”).

“w”: moving camera forward along “dir”.

“s”: moving camera backward along “dir”.

“a”: moving camera leftward along “right”.

“d”: moving camera rightward along “right”.

“u”: moving camera upward along “up”.

“n”: moving camera downward along “up”.

I use the direction keyboard to control camera rotations.

“⬆”: rotate counter-clockwise along “right”, making direction more upper.

“⬇”: rotate clockwise along “right”, making direction more down.

“<-”: rotate counter-clockwise along “up”, making direction more left.

“->”: rotate clockwise along “up”, making direction more right.

I also have animation control over “debug” camera. Using “[Space]” to resume/pause the playing animation.

*Command Line Arguments* :

--scene scene.s72 (required) -- specifies the scene (in .s72 format) to view.--camera name (optional) -- view the scene through the camera named name. If such a camera doesn't exist in the scene, abort.--physical-device name (optional) -- use the physical device whose name matches. If such a device does not exist, abort.

--list-physical-devices (optional) -- show all physical devices names.

--drawing-size w h (optional) -- set the initial size of the drawable part of the window in physical pixels. If not specified, using default 800 x 600 window size.--culling none | frustum | (optional) -- sets the culling mode. --headless events (optional) -- if specified, run in headless mode (no windowing system connection).

*Support Code for Math*

Using template to write math lib, including Vec <N, T> with size N and type T; Mat<N, T> of a matrix of size N x N. I don’t support any non-square matrix since I am not using them. Usual cases of Vec <3, float> and Mat <4, float> is included as “using” so they can be written as Vec3f and Mat44f. Besides, I also implemented some basic requirements to generate rotation matrix, translation matrix, inverse, scale matrix, … so on, not to mention the basic matrix-vector-multiplication.

*Support Code for Vulkan* :

Vulkan structure is much similar to that in the tutorial. The structure is basically the same with the separation of each part in each specific source files (i.e. graphic\_pipeline’s management are in its own source file) for the simplicity of management. Resources creation and destroy are managed together in scene\_viewer.cpp, where I unified the management of resources.

*Scene Loading (Mesh)* :

Starts with JSON parsing, it’s a simple approach that I remove all the inner space and “\t”, so the information is dense. Then, I have 2 kinds of recursion function “BuildObject” and “BuildVector”. If any element starts with ‘[‘, then it must be and vector, so called “BuildVector”, otherwise I call it “BuildObject” (notice any element starts with ‘{‘, should also starts a call to “Object”). This mcjp, JSON Parser, could be improved by using shared\_ptr for better resources management, currently it requires explicitly cleanup.

Detailed Object class can be seen in file scene\_config.hpp where I use to manage all .s72 file Objects’ classes (Mesh, Node, Driver, …). Each Mesh contains the basic positions, normal, color information. Besides, it should also maintain some meta-data that contains inner data itself. Here I use shared\_ptr to manage resources. Data is stored mesh-uniquely, so we only would have one copy of each kind of mesh in vertex buffer. (BTW, vertex buffer’s creation/allocation is after JSON parsing, so that we can know the total number of unique vertex). Also, along with mesh parsing, I also generate its bounding sphere with it. I tried 2 methods here one is the Ritter’s algorithm (implemented in main branch), the result is generally good. And a second approach is using min/max of x/y/z to generate a bounding box(aligned with coordinates), then use the diagonal to generate a big bounding sphere. The second approach is simple and easy-understanding, and it also works with some culling stage.

*Drawing Scene* :

Note we have our scene graph, thus we are actually drawing possibly-visible instances of each mesh. So, I decide to have one copy of each mesh in Vertex Buffer, and assign different InstanceIndex to those in current view. Thus, in each frame, I would have a 2-dimensional vector, where the first dimension denotes the mesh-id and the second dimension denotes the instance-id. The vector contains the current transformation matrix (consists of translation \* rotation \* scale) of each instance. The progress is in each frame, we start from the roots, record down “*currentTransformation* ”, pass it down to each child in the scene graph (a DFS approach). While dfs’d down, “*currentTransformation* ” is left-multiplied with parent’s transformation, then pass it down … until we have some meshes. Based on those mesh’s Id, push\_back them into correct index of that 2D vector.

Then, we use that 2D vector, contains each transformation matrix of each instance, which is also, the model matrix of each instance (with specific instanceId), and I use that to update the uniform buffer instead, in vertex shading.

*Handling Interactive Camera* :

This is associated with another 2 matrices in viewing, which are View Matrix and Projection Matrix. But the camera’s parameters like vfov or aspect ratio will not change during our viewing, so Projection Matrix is fixed, so we only need to care about View Matrix, which is based on camera’s position, watching direction and up direction. Since we have camera controls from user’s input, each frame we also require current View Matrix update in image\_view progress.

*Frustum Culling* :

Frustum Culling involves 2 parts, camera and objects. Camera’s movement will change the surrounding frustum in view, thus we need to update 7 points of frustum in world-space after we modify of our camera.

[frustum image]

Where these points can be easily calculated based on origin\_near, origin\_far, dx\_near, dx\_far, dy\_near and dy\_far. Remember they are already in world-space, so we do not need to further apply the (VP) inverse matrix on them.

Then goes with objects. During the load-time, I generate a bounding sphere for each unique mesh. I tried two methods with it, the first is Ritter’s algorithm, which is around O(3\*n) time to generate a kinda small enough bounding sphere, on its record, the average size is around 1.5 size bigger. Another is O(1\*n) time to generate a bounding cuboid using min/max of x/y/z. Then using the diagonal to generate a bounding sphere. It’s fast but might be a little big. Then during the transformation (also applied with the animation transformation), I record the max absolute value of scales in x/y/z, then apply it to the radius as a new bounding sphere. This is fast for each frame update, but has one drawback that, once there are some serial scale-transformations in the animation, the sphere is getting larger and larger, thus, making bounding sphere useless. But applying it on translation and rotation, the results are generally good.

[image of sphere and a plane]

Then goes to how I decide where culling or not. Basically, for those 6-plane of a frustum, they are updated/generated together with a normal and a distance from origin(world space). Then, when we go through each instances along the DFS over scene graph(in each frame time), (which is necessary for getting their transformation matrix used in uniform buffers) we can apply the transformation on the center of that bounding sphere and then test whether the sphere is visible. Visible here can be easily test the center of bounding sphere to each plane of the frustum. As long as the distance + radius is greater than 0, this instance is possibly visible. (Rejecting all false-negative).

*Animating the Scene* :

This is applied easier now, since we have built up the structure of scene graph DFS, so if these nodes are applied with animation, we can simply replace those animation matrices to their original ones.