

Computer Games Development CW208

Technical Design Document

Year IV

# Robin Meyler

C00231699

04/05/2020

**Contents**

Rendering Pipeline 3

Compute Pipeline 5

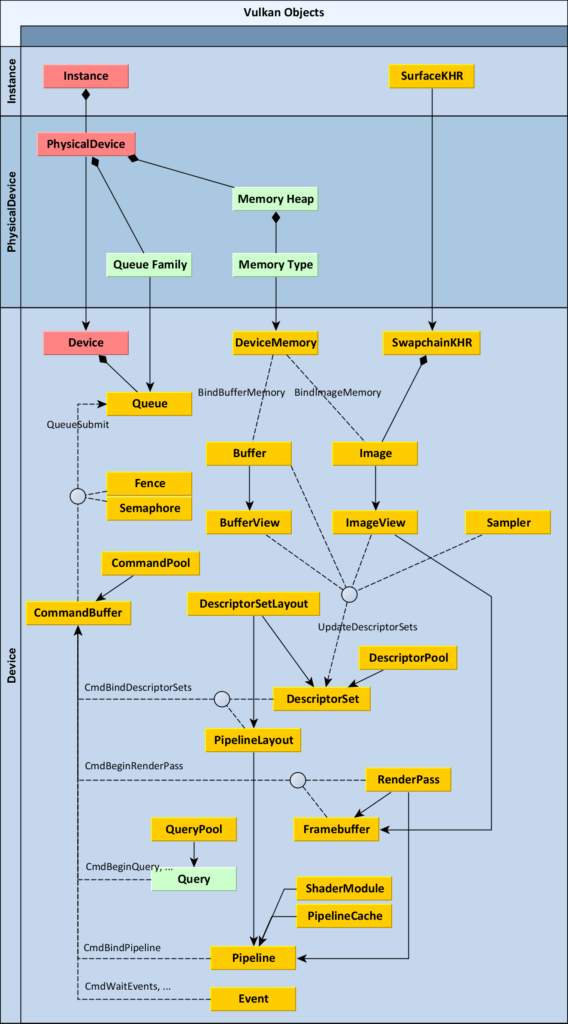
A\* 6

Priority Queue 7

Features 8

CRC Cards 12

References 15



**Rendering**

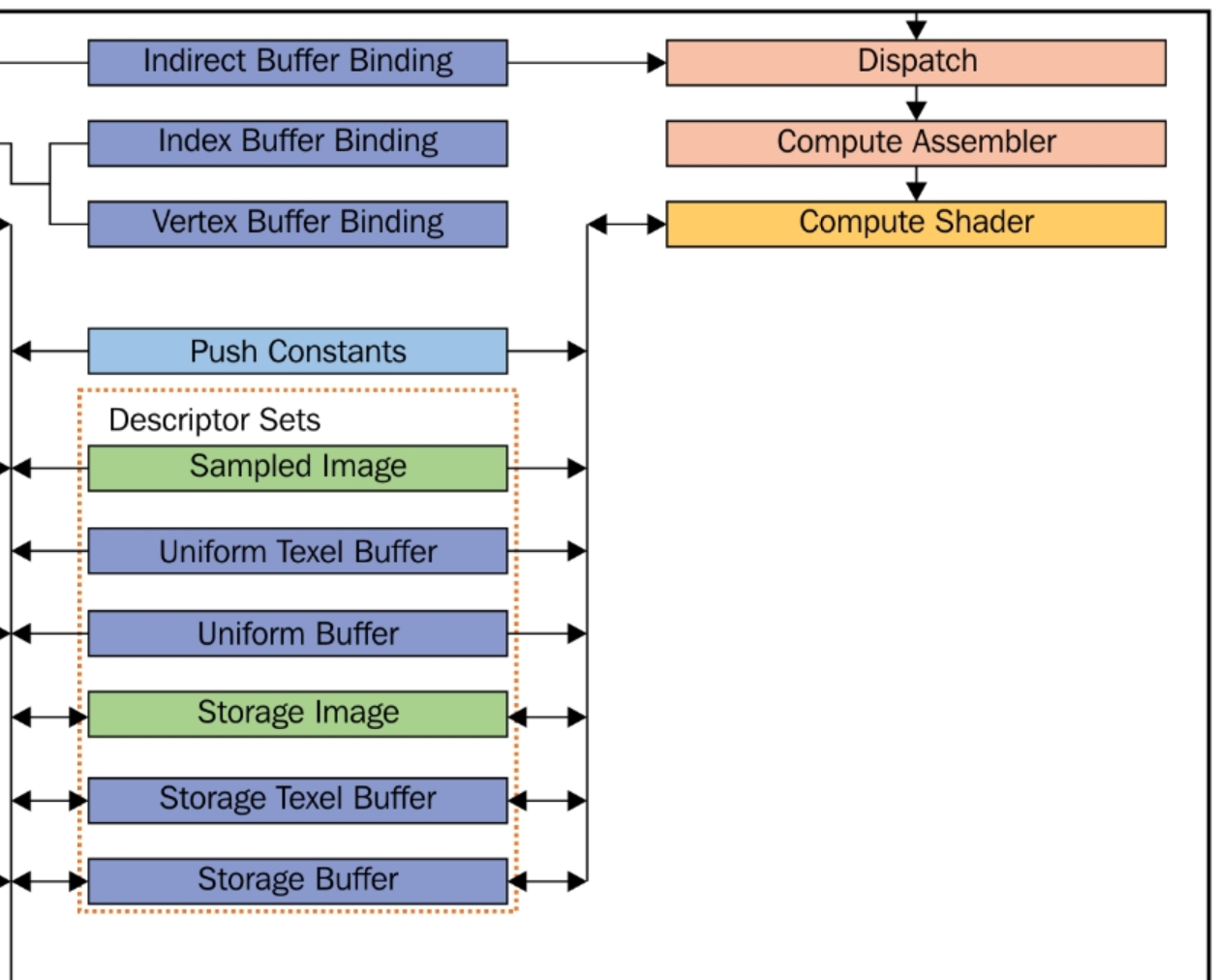
For the Rendering pipeline for this application, I will be using the Vulkan API, Vulkan was created by the creators of OpenGL as their newer more up to date graphics API. The main ideas of Vulkan was to unify the Rendering and Compute functionality into 1 api and to support use of multi core computation on modern hardware. The main design principle of the Vulkan API was to allow extremely explicit and verbose control of the pipeline to the user. OpenGL managed a load of the complex GPU functionality, memory management and made loads of assumptions on how to setup your program. Vulkan does none of this, Vulkan requires you to explicitly declare and configure every single part of the pipeline and if it is not set by you, it won’t be set. This design is to give complete control to the developer with the philosophy that the developer knows better how to optimize their program than a generic setting that OpenGL might employ.

The first part of the pipeline is to create a Vulkan instance, a base for your project to exist on, the instance can enable some instance level extensions available from your driver. Since Vulkan is completely agnostic, next we need to create a specific surface to draw upon. Extensions from the Kronos group themselves mixed with in-built functions from GLFW make this an easy step. Next we choose a physical device, some machines may have more than 1. You can employ any sort of criteria here as you see fit but it’s most important to ensure the device has Graphics and Compute support as well as the ability to support Presentation. Most modern desktops will have this functionality. Next we create a Vulkan Device, this is where we set device level extensions as well as query for and choose queues and queue families. Commands in Vulkan are recorded in a common buffer and during drawing submitted to the GPU, these commands are offered from Queues which have a queue family that are available to your GPU. We declare the Queue(s) we need here.

Next we have to set up frames to draw for the program. For this, khronos offer an extension called a Swapchain for windows machines. The Swapchain is essentially a set of frames that are cycled through drawing to 1 while presenting another. The Swap Chain is used for double and triple buffering. Our program will use triple buffering so the swapchain will have 3 images in it, each image has an image view that declares what part of the image to view, this is declared next and finally every image and image view requires a framebuffer to show the information to draw. Next we create a Render pass which tells the program what to do with the information for these sets of commands and is used in the command buffers during draw time, it declares subpasses and information on whether we care what happens to info after a render or before it. Next we create a Graphics pipeline that configures every part of the pipeline. It's also during pipeline creation that we create shader modules from our shader scripts and load them to the GPU.

Next we create all the buffers we need such as vertex buffers, index buffers, uniform buffers, storage buffers and instance buffers. We declare the buffers then query for information about the buffers, we then use this information to allocate GPU memory for the buffers and then copy the buffers to device local memory on the GPU which is the most performant memory. The basic information about the vertex buffers (Cube in our case) is set with binding information and attribute information which tells it which layout variables to use in the shader when processing it, and the stride between vertices for the memory layout. We then set up descriptor sets to tell the program how to deal with uniform buffers information as we update it within the shaders.

Finally we create command buffer objects which are essentially a set of commands that we can recall each frame to bind buffers, pipelines, choose a render pass and draw by submitting the graphics queue queried earlier. command buffers are allocated from command pools and then declared during setup. Lastly we create synchronization semaphores and fences to make sure we have control of operations on a CPU to GPU bases as well as GPU to GPU basis to ensure we don’t try and present an image that is not finished drawing but also begin drawing on a new swapchain image before the last frame is fully presented.

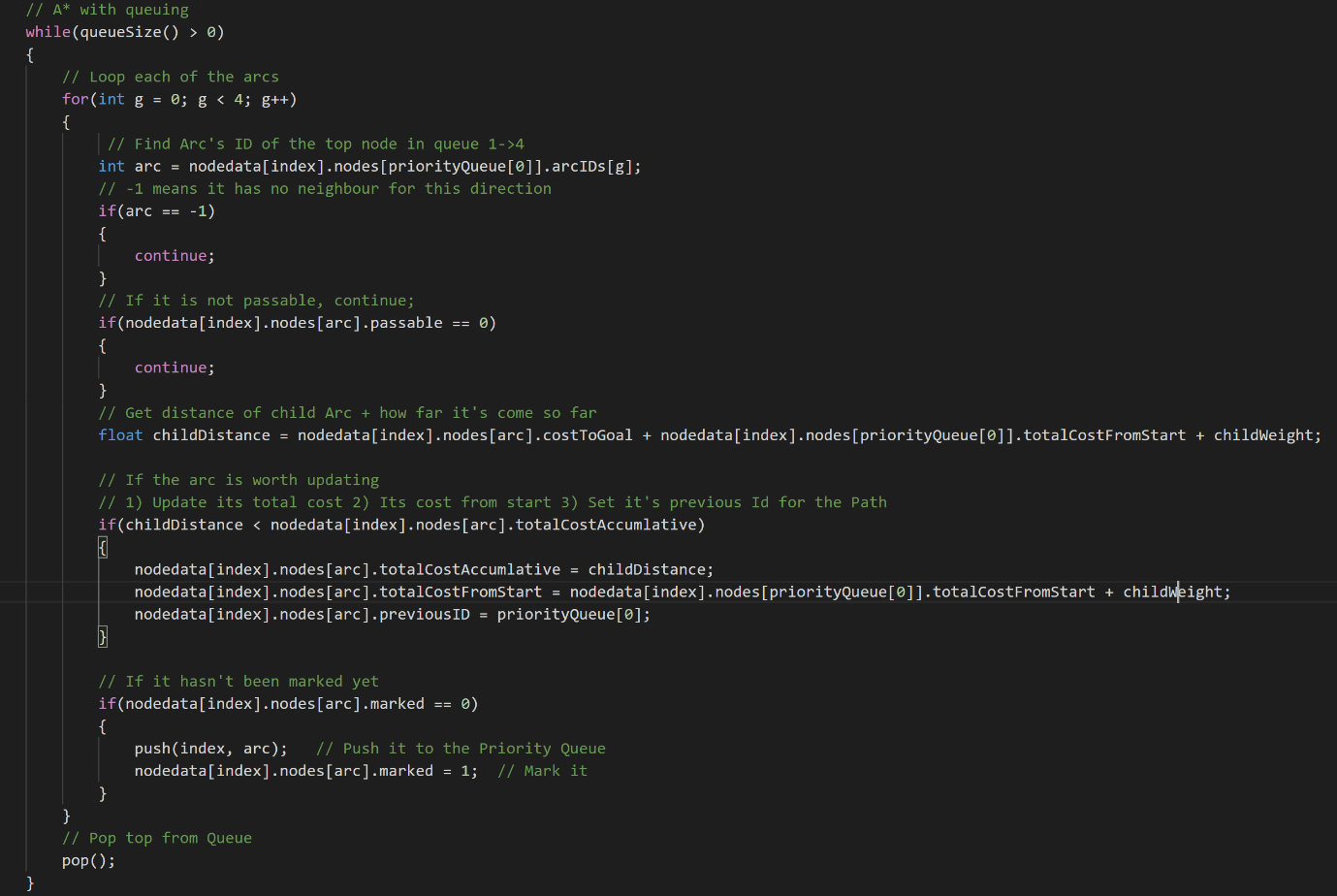
****

**Compute**

The Compute pipeline in Vulkan is a separate mechanism of interacting with the GPU. The compute pipeline has the ability to skip many steps usually associated with a GPU call such as Vertex processing, rasterizing, tessellation etc. The Compute simply prepares information for a compute shader with resources of either a Storage buffer or Image to interact with and process. Uniform buffers can also be used with the Compute shader. In our case, the appropriate buffer is a storage buffer as we can construct Structs with the information we need, allocate memory for it and then map it to the GPU storage buffer. Then, upon Compute dispatch, the invocation index will direct point to one stride of memory and since I used Vulkan to create this memory to an exact memory size, the values match up as expected. You can then choose how many workgroups you wish to dispatch and how many per workgroup but for our case we will be sending 1 large workgroup to improve performance.

The Compute Pipeline requires a certain amount of extra steps from our Rendering Pipeline to allow us to use both mechanisms in one program. Firstly the descriptor sets that we must declare from Descriptor pools must be altered. Descriptor sets describe to the GPU how to interpret uniform data. We need to update the sets to also expect compute information. Next we need to create the Storage buffers for both the Node data and the Paths array we will fill and use after execution. Next we need to allocate memory on the GPU for our newly created buffers and bind them. Then we create a compute pipeline separate from our rendering one and set it up with the correct configurations to expect the compute operations. This configuration allows us to skip a lot of settings needed for normal rendering. Finally we create a command buffer that begins operations, dispatches the compute shader with the desired amount of workgroups and waits for the GPU to become idle, signalling that the operation has been finished. After recording execution time, we then access the memory with Vulkan mapping and assign the returning paths to CPU memory for use in our program.

**A\***

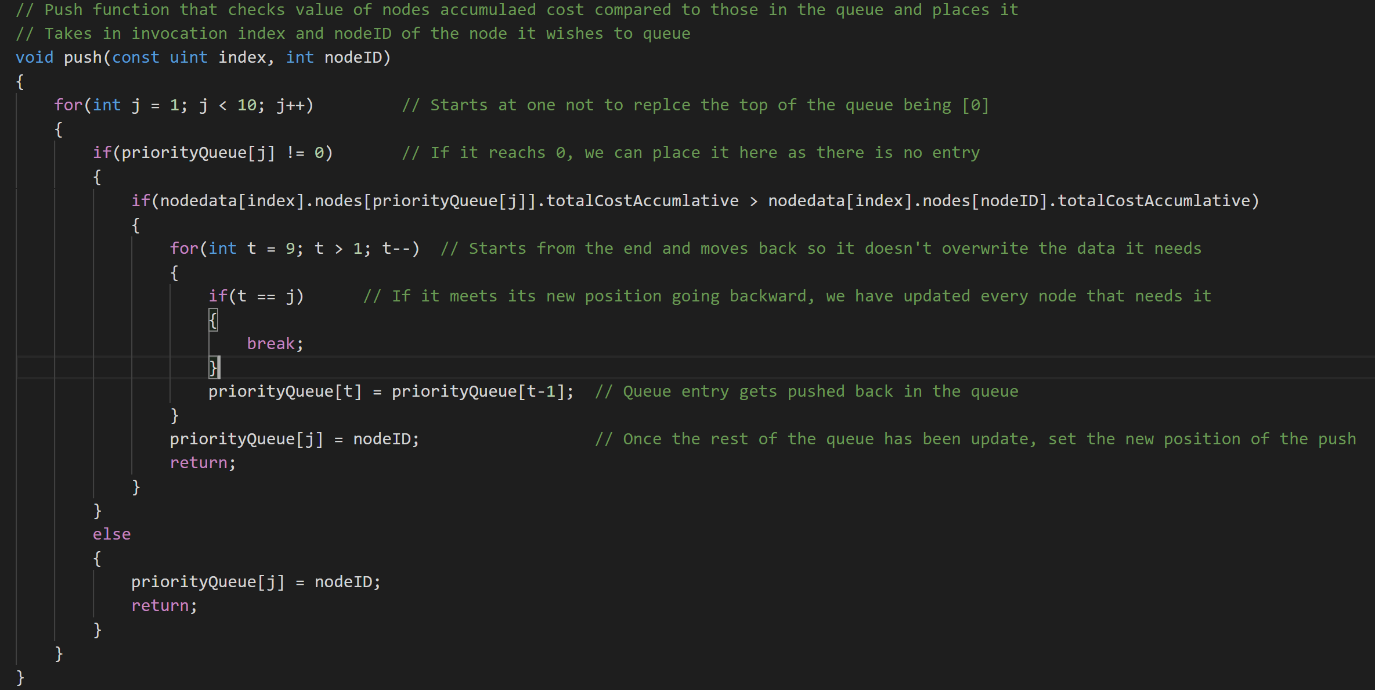
****

Pathfinding is a method of determining the fastest route for a moving object to another position in an obstacle rich environment. A\* is one of the fastest versions of pathfinding that uses nodes, distance, a cost heuristic and a priority queue to determine the fastest route to a goal position. It uses a total accumulated cost to determine the fewest amount of nodes it needs to process by ordering the nodes to check next based on this heuristic.

To use the GPU to execute the A\* pathfinding for several agents, I needed to interact directly with the GPU to get it to compute tasks, for this I use Compute shaders, based in GLSL. After building the compute pipeline, it can use this compute shader to execute on storage buffers to determine their paths and fill a second storage buffer with the results.

I create the A\* loop seen above, which uses a custom priority queue, the invocation index and the set of nodes to determine the path for this agent. This shader gets computed in parallel in warps of 32 threads of paths being filled for the starting positions passed in for each agent. All of the agents have their own node information because cross invocation communication is difficult and comes with a performance cost as it is generally advised to keep all information to do with an invocation local to the shader. The script checks for the queue size, checks the nodes children 1 by 1 to see if they are passable, what their cost is and if they are marked, if they have not already been set, they are updated and pushed to the priority queue.

**Priority Queue**



One main feature of the A\* pathfinding algorithm is the priority queue it employs to determine the best node to process in what order based on the total accumulative cost it gathers while the algorithm goes. This accumulated cost is based on the cost from goal mixed with how far the path finding thread has gone from the start plus the weight to that child.

Since I have no access to a standard library’s priority queue, I’ve had to create my own inside the shader for each invocation to have access to its own queue while processing the path. Instead of pointers, this queue uses an INT array, which is the same memory footprint and uses the NodeID of the specific node to process. It can use this idea to access the node’s information to determine where in the queue it should place. PQ[0] is considered the top of the queue and is the node processed in the main A\* loop.

The Queue consists of 3 functions, Size() which checks for node IDs of -1, which indicate no entry. Any entry that is not -1 means an entry in the queue this is counted and returns, it is used in the main While loop to determine if A\* should continue or not. The second function is called pop(). This function takes every entry in the loop and moves in up one, then changes the last member of the queue to -1. ( PQ[1] - > PQ[0], PQ[2] - > PQ[1]) The last function is the one shown in the image above. This function takes the invocations index and the new NodeID to push to the queue. It starts at the second entry since PQ[0] is the top and being processed, it checks if the entry is unfilled and if it isn’t filled, place the new node here and break; If they are filled, check if the node in this position’s accumulated value is greater than the new node’s value. If we find a value greater than our new node, we start at the end of the queue and move every node back 1 place till we meet our new slot. We place the node in this newly freed slot and break from the queue.

# Features

## Feature: Rendering 3D

**Tasks:**

1. Create an Instance, Device and choose a physical GPU to use.
2. Create a window to draw to with GLFW.
3. Create a Swapchain for multiple image buffers, an image for each, an image View for each image and a frame buffer for each image view.
4. Create a Renderpass that specifies information per pass and per subpass about how to interpret information.
5. Create a Graphics pipeline that configures each part of the pipeline from vertex input to rasterization. Configure the vertex’s input data.
6. Create Command buffers from command pools and determine the logical calls and their order before drawing to be reused in the program
7. Setup descriptor sets from descriptor pools to define information about uniform buffers.
8. Create a Cube object, vertex buffers and index buffers then tell the GPU how to handle it.
9. Create synchronization semaphores and fences to manage frames, rendering and presenting.

## Feature: Compute Functionality

**Tasks:**

1. Create structs to hold storage buffer information.
2. Create the storage buffers, allocate memory for them then copy the memory onto optimal GPU memory for future execution.
3. Create descriptor sets for the compute shader.
4. Create a Compute pipeline and configure it to expect the compute operations.
5. Create compute command buffers to be called in the program to dispatch the compute shader.
6. Map info into the storage buffer.
7. Reclaim this memory from the GPU after execution.

## Feature: GLSL A\* compute Shader

**Tasks:**

1. Create a Vertex Shader to process vertices using MVP from the uniform buffers and pass them to the rasterizer.
2. Create a Fragment Shader to draw to the screen with desired information.
3. Create a Compute Shader for A\* and create structs and initialize it with storage buffers.
4. Decide on the local size of workgroups.
5. Locate the Global ID index of the specific invocation to be used as reference.
6. Create own in shader priority queue for the A\* computation using an array of ints.
7. Create/Port standard A\* pathfinding into the shader using the storage buffer for information and the priority queue for ordering.
8. Fill a second storage buffer of pathIDs after computation of the fastest path for each invocation for the agents on the CPU side.

## Feature: Pathfinding

**Tasks:**

1. Create CPU side structs to match the compute shader scripts exactly, align for any memory issues.
2. Setup node information for a grid with neighbours stored as node IDs and weight between neighbours being the same for every neighbour, -1 for edges
3. Ensure memory is declared for the information in the storage buffers.
4. Create pathfinding agents with start positions.
5. Create a goal to pathfind to and update the pathfinding agents.
6. Calculate the cost to goal and then map the agent memory to the storage buffer memory on the GPU for dispatch later.
7. Dispatch the A\* compute shader upon user input.
8. Claim the resulting path data by mapping back to the GPU memory.

## Feature: Camera

**Tasks:**

1. Ensure Descriptor sets from a descriptor pool were allocated for uniform buffers.
2. Create a Uniform buffer from a struct of 3 matrices, Model, View, Projection.
3. Use GLM to create each MVP to map to the uniform buffer.
4. Update the buffer with the new information by mapping to the GPU memory directly.
5. Use user input to move the position of the camera and the position of where the camera is aimed to scan the grid

**Feature**:Grid Creation

**Tasks:**

1. Create vertex buffers equal to the number of required obstacles, walls and moving agents in the demo.
2. Adjust the position of the cubes to match a grid layout.
3. Fill each created buffer object with cube information and create an index buffer to lessen the amount of vertices processed per cube.
4. Copy the vertex buffer information onto the GPU and tell the rendering pipeline what size buffer to expect, how to use it and what layout variables to use when processing them.

**Feature**:Instancing

**Tasks:**

1. Create a second graphics pipeline to be almost the same but expect the vertex input data to be instanced.
2. Create new descriptor sets to expect information about how each instance data should differ from the base.
3. Create an instance buffer that has information on the position of each cube to be instanced and map it to the GPU.
4. Call the instanced buffer with 1 cube vertex buffer and tell it to draw all the cubes in the program with 1 call.

## Feature: Timing/Testing

**Tasks:**

1. Include Chrono library and learn how to use its clock system for precise time taking.
2. Start a timer just before dispatching the A\* Compute shader and get the time again as soon as the GPU goes idle to determine execution time.
3. Open a file of a desired name and write out the time gathered in microseconds into this file after converting it into seconds.
4. Repeat this test for many Grid sizes and Agent counts writing out to new files, many times each.
5. Read from the files after many times were taken, combine the results and get an average for each setup.
6. Compare results.

## Feature: IMGUI

**Tasks:**

1. Setup initialization information for IMGUI and give it a pointer to the GLFW window instance.
2. Create a new descriptor sets for the UI
3. Create a new render pass to be called after the rendering render pass to ensure it draws above the image.
4. Init IMGUI with all the pipeline and Vulkan information and upload fonts.
5. Create a new framebuffer for each frame in the program to match the current image view so information can be shown in the same view but from a different memory set.
6. Render the UI.
7. Recreate/update the render pass.

# CRC Cards

|  |  |
| --- | --- |
| Class Name : Vertex 3D | |
| Subclasses : | |
| Superclasses : | |
| Responsibilities  **Variables**   * Vector 3: Positions * Vector 3: Colour   **Methods**   * Get Binding Description (Vulkan struct that declares binding, stride) * Get Attribute descriptions (Vulkan object that declares shader binding, location, format and offset) | Collaborators |

|  |  |
| --- | --- |
| Class Name : Cube | |
| Subclasses : | |
| Superclasses : | |
| Responsibilities  **Variables**   * Container of Vertex 3D: vertices * Container of Ints: indices   **Methods**   * Constructor that takes position * Update colour of each vertex * Update position of each vertex | Collaborators   * Vertex 3D |

|  |  |
| --- | --- |
| Class Name : Renderer | |
| Subclasses : | |
| Superclasses : | |
| Responsibilities  **Variables**   * Vulkan Instance * Vulkan Surface * Vulkan Swapchain * Container of Vulkan Images * Container of Vulkan Image Views * Vulkan Format * Vulkan Extent * Vulkan Physical Device * Vulkan Device * 2 Vulkan Queues, Graphics + Present * Vulkan Render Pass * Vulkan Pipeline Layout for Rendering * Vulkan Pipeline for Rendering * Container of Vulkan Framebuffers * Vulkan Command Pool * Container of Vulkan Command buffers * 2 Containers of Vulkan Semaphores for image management * 2 Containers of Vulkan Fences for image management * Size\_t for current image index * Container of Vulkan buffer for Vertex Buffer Objects * Vulkan buffer of Index Buffer Object * Container of Vulkan buffers for Uniform Buffer Objects * Vulkan buffer for Staging buffer * Vulkan buffer for * 2 Vulkan descriptor set layout * 2 Vulkan descriptor Pool * Container of Vulkan Descriptor Set * Vulkan Pipeline Layout for Compute * Vulkan Pipeline for Compute * Vulkan Command buffer for Compute * Several Vulkan Device Memory objects * Container of Path structs * Container of NodeData structs   **Methods**   * Create Instance * Create Surface * Choose GPU device * Create Logical Device * Create Swap Chain * Create Image views * Create Render Pass * Create Graphics Pipeline * Create Compute Pipeline * Create Descriptor Set Layout * Create Frame Buffers * Create Command Pool * Create Buffer Objects * Create Vertex Buffer * Create Index buffer * Create Storage Buffer * Create Descriptor Pool * Create Descriptor Sets * Create Command Buffers * Create Sync Objects * Setup Vulkan – which calls all the above functions and is called from main loop * Draw – Gets the next image in the swap chain for rendering, uses fences to manage when to draw * Clean Up – Destroys everything at end of program * Add Nodes * Add VBOs – Vertex Buffer Objects * Allocate Buffer Memory – declares memory on GPU to be used by the program of specific size * Set Starts - starting positions of Agent cubes * Update buffer | Collaborators   * Cube Struct * NodeData Struct * Path struct |

# References

|  |  |  |
| --- | --- | --- |
| Book | (AM. 2018) | Akeine-Moller A. Haines E. Hoffman N. Pesce A. Iwanicki M. Hillaire S. (2018), Fourth Edition, *Real Time Rendering.*  CRC Press |