Lappeenrannan teknillinen yliopisto
School of Engineering Science
Tietotekniikan koulutusohjelma

Kandidaatintyö

Henrik Valve

PROTOTYPE IMPLEMENTATION OF PERIMETRY ON VIRTUAL REALITY HEADSET

Työn tarkastaja: Professori Ajantha Dahanayake

Työn ohjaaja: Professori Ajantha Dahanayake

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
School of Engineering Science
Tietotekniikan koulutusohjelma

Henrik Valve

Prototyyppi toteutus näkökentän testaukseen virtuaali laseilla

Kandidaatintyö

2019

31 sivua, 3 kuvaa, 1 taulukko

Työn tarkastaja: Professori Ajantha Dahanayake

Hakusanat: prototyyppi, näkökenttä, näkökentän testaus, VR-lasit,
Keywords: prototype, visual field, perimetry, VR-headset

Työssä tuotetaan prototyyppiohjelma, joka pystyy tekemään näkökentän tutkimisen VR-lasien avulla. Toteutus toimii Windows käyttöjärjestelmässä, johon on asennuttu Vulkan ajuri ja tietokoneessa on VR-laseja tukeva grafiikkakortti. Siirrettävyys eri käyttöjärjestelmien ja VR-lasien välillä otettiin huomioon suunnittelussa. Ohjelma on koodattu C++:lla. Työssä ei puhuta graafisesta käyttöliittymästä tai potilastietojen tallennuksesta tarkasti vaan keskitytään tausta puoleen toteutukseen. Työssä puhutaan syvällisesti miten yksinkertainen Full Threshold algoritmi ja miten piirtäminen Vulkanilla tehdään.

ABSTRACT

Lappeenranta University of Technology

School of Engineering Science

Degree Program in Computer Science

Henrik Valve

Prototype implementation of perimetry on virtual reality headset

Bachelor's Thesis

31 pages, 3 figures, 1 table

Examiners: Professori Ajantha Dahanayake

Keywords: prototype, visual field, perimetry, VR-headset

In this work we provide prototype of a program that can make perimetry (visual field test) on VR-headset. Implementation works on Windows operating system, which has Vulkan drive and computer has graphical processing unit which supports VR-headsets. Portability to other operation system or VR-headsets was taken into account in the design process. Program is coded in C++. This work does not speak about graphical user interface nor patient's information storage in detail rather focuses the the back end of the implementation. We will discuss in detail how to implement simple Full Threshold algorithm, how the rendering was done in Vulkan.

TABLE OF CONTENTS

1 INTRODUCTION....3

2 OTHER PAPERS ON SUBJECT MATTER OR RELATED TO IMPLEMENTATION....5

3 VISUAL FIELD TEST PROGRAM....7

3.1 Overall architecture....7
3.2 Main in detail....12
3.3 Automatic perimetry algorithm module in detail....13
3.4 Model operations in detail....18
3.5 Rendering in detail....20

4 IN CONCLUSION....26

REFERENCES....27

ABBREVIATION LIST

API	Application Programming Interface
GUI	Graphical User Interface
GPU	Graphics Processing Unit
SITA	Swedish Interactive Thresholding Algorithm
VR	Virtual Reality

1 INTRODUCTION

Do test patient's visual field ophthalmologist use automated visual field test or automated perimetry machine. This testing gives ophthalmologist reliable way to diagnose glaucoma and other visual field diseases. Test can be static or kinetic. In kinetic testing patient is asked to fixate front of them with one eye and then something is moved to their visual field. Patient should tell when object enters into their vision. Kinetic testing is usually performed to children manually. Static testing involves patient again fixating to point with one eye. However points are shown as flashing light rather then moved to patient field. This is what perimetry machines do. Patient is sitting front of flashing light panel on eye closed and presses a button when she sees stimulus. Patient is shown light at same points on different intensity levels so that ophthalmologist would know the threshold of patient sees stimulus where she doesn't. Since test involves patient fixating on same point long periods perimetry algorithms are develop to make test fast as possible. Efficiency also helps with rapid testing of population.

Normal human monocular (one eye) visual field is roughly 60 degrees superiorly (up), 70 degrees inferiorly (down), 60 degrees nasally (towards the nose) and 100 degrees temporally (away from the nose) from centre of fixation [1]. Human vision has blind spot due how optic nerves are "wired" at the eye. This blind is roughly located 1.5 degrees inferiorly and 15 degrees temporally. It's size is 7.5 degrees high and 5.5 degrees wide [1]. Sensitivity threshold reacts to forms hill of vision where hill is most sensitive at the fixation and then goes down further (in degrees) from fixation we get.
Hills high goes down over age [2]. Intensity of the light is represented by decibels. However there is no standard how these decibels map to luminous intensity units like candela. Usually zero decibels is highest intensity but how much that is depends on machine [2]. We use zero to 40 range on decibels.

We provide implementation of automated perimetry machine's back end on virtual reality (VR) headset written in C++. There are multiple algorithms that can be perform the test. Our implementation implements full threshold but is designed to allow multiple ones. Design has some flexibility when it comes to size of the stimulus and fixation. Also common 30-2 grid

pattern is given but no other grid patterns are supported at the moment. These are discussed more up upcoming sections.

Potential advantages of this software would allow normal people have better access to self diagnose first test on them self and other wise make it easier to get people testing there eyes. Other potential is that VR in future could be far more cheaper method then current perimetry machines.

2 OTHER PAPERS ON SUBJECT MATTER OR RELATED TO IMPLEMENTATION

Program's that can do perimetry in VR headsets isn't new idea. Two studies develop a prototype for perimetry on VR headsets [3] and [4]. Studies concluded that their device was comparable to Humphrey Field Analyser which is golden standard of the perimetry devices. Difference between studies is that first one used custom made headset and second one used Trust EXOS 3D VR which uses mobile phone as the screen. First study included eye tracking. Nether study did focus on too much on there actual implementation details.

There is multiple algorithms to perform perimetry. State of the art is SITA (Swedish interactive thresholding algorithm) first discussed in [5]. However we implemented older Full Threshold algorithm [1] (or normal strategy in [2]) because SITA on certain conditions falls back to Full Threshold and SITA's post-processing phase is ill-defined in the paper [5]. Full threshold checks multiple locations of visual field. Usually these locations are intersections of the grid some degrees apart each other. On every location staircase procedure is used where first stimuli if seen will cause intensity of next stimuli on location to be dropped by 4 decibels. Intensity will continue to drop if patient still sees the stimuli until stimuli isn't seen which point stimuli is increased in 2 decibels increments to before patient sees stimuli again. This will be the threshold of that location. If first stimulus isn't seen then directions of the intensity change between the steps is reversed.

There are multiple test patterns perimetry can be done. There is no one pattern fit for all rather there is ones that focus on different areas or trade accuracy to speed. We implement 30-2 grid which is has 76 points in circle separated by 6 degrees from each other. Most closes points to fixation are at corners of 3 degree rectangle where fixation point is centre off. This grid is mean to be used to detect glaucoma [2].

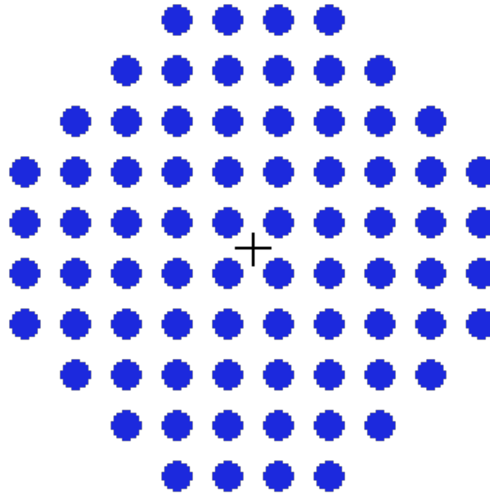

Image 1. 30-2 grid pattern diagram. Middle cross hair is fixation point.

Fixation point in our device is rendered as circle but as [2] points out some individual may not see such point. We didn't program interface for changing fixation target rendering model but such interface is not hard to add what we discuss here.

Program was first develop for HTC Vive headset. Then development sifted to Fove headset for it’s eye tracking ability. However eye tracking capabilities weren't implemented.

3 VISUAL FIELD TEST PROGRAM

3.1 Overall architecture

Program is divided to modules that handles certain abstraction or data structures management. Following table has all of the modules listed. Almost every module is depended on other module. Modules *VrVisualFieldGui* and *DatabaseIO* aren't part what we call back-end.

Table 1. Modules of the program and there task.

Module name	**Purpose**	**Functions**
VrVisualFieldTestMain	Start of the program. Calls initialization functions and start threads.	main
VrHeadset	Abstraction for VR headset API's.	vrHeadsetDetect vrHeadsetInit vrHeadsetDeinit vrHeadsetGetInputMethod vrHeadsetGetGazeVector vrHeadsetSubmit
RenderingEngine	Abstraction for talking GPU.	renderingLoop renderingInit renderingDeinit renderingScaleStimuli renderingScaleFixation renderingStimuli renderingResetStimuli
VrVisualFieldTestGui	Abstraction for GUI handling.	guiIsFinishedCallbackForPerimetry guiInit guiStartEventLoop guiAddEvent
AutomaticPerimetryAlgorithms	Houses all of the perimetry algorithms and thread that waits command to execute one of the perimetry algorithms.	apFullThreshold apSignalWaiter apSignalToWaiter apSignalCondition apGetEye apCancelPerimetry
PatientInput	Takes in patient's input on separate thread and adds it to model.	patientInputLoop patientInputStart patientInputEnds
VisualFieldModel	Has functions to allocate visual field model and to store stimuli data turning perimetry.	modelAllocate modelAllocatePoints modelGetPoint modelCommitPoint modelGetCommitLock modelReleaseCommitLock

		modelPushCommit modelDelete modelAddReaction modelIsSecondInitReady
TimeOp	Support library for handling *struct timespec*.	timeOpAddConstMs timeOpSub timeOpCmp timeOpConstMsEpochCmp timeOpLong
DatabaseIO	Abstraction for saving result of the perimetry to database.	databaseCreate databaseSaveModel databaseClose

Program is designed to have four threads:

- thread that runs perimetry algorithm when ordered
- thread that handled rendering to headset
- thread that takes in patient's input
- thread that handles events for GUI (graphical user interface) for technician

These are done in separate threads because they have complete different timers and putting them on same thread (or combining) could cause slowdown on some critical moment.

To "hide" the abstraction program has multiple handle types:

- Structure *VrDevice* is data structure for VR headsets
- Structure *WindowHandle* is gui abstraction
- Structure *Database* is database abstraction.

VrDevice is only structure that is designed to be class of object oriented programming. Different headsets API's have different implementations of *VrHeadsetInit*, *VrHeadsetDeinit*, *VrHeadsetGetInputMethod*, *VrHeadsetGetGazeVector*, and *VrHeadsetSubmit*. Function *VrHeadsetDetect* is designed to detect different headset and then start initializing right headset.

Other data structures in the program are:

- Enumaration *VrEye* which is used by AutomaticPerimetryAlgorithms to tell which eye is under testing or wanted to be tested

- Enumaration *InputMethod* is used by *VrHeadsetGetInputMethod* as bit field return value to tell caller what input methods patient can used (*KEYBOARD*, *CONTROLLER*, *GAZE*)
- Structures to create visual field model (*Model, Point, ReactionDataNode, ReactionDataMemory*). Image 2 is UML "class" diagram for the of these structures.

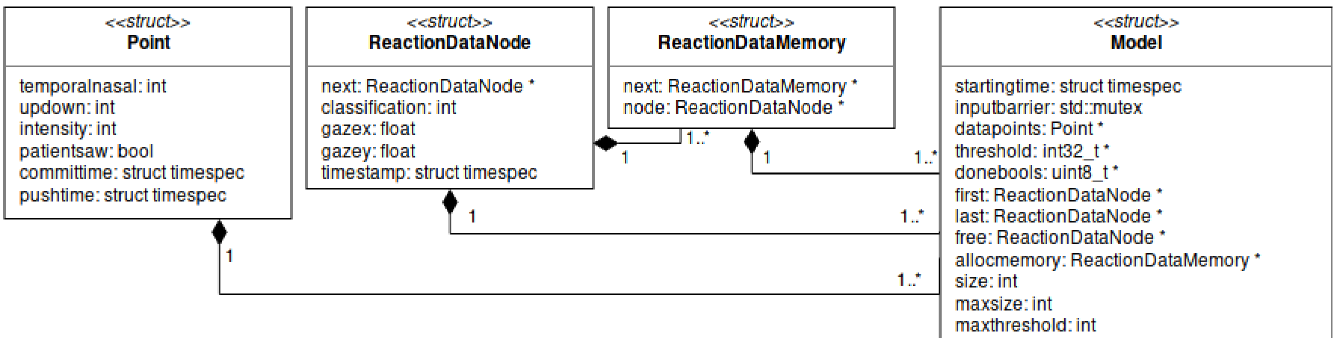

Image 2. Class diagram of the Visual Field Model.

Structure *Model* is the base of the visual field model having general information and pointers to information collected during the testing. Member *datapoints* is array of stimuli points shown during the test. Member *maxsize* is how many big array *datapoints* is and member *size* is incrementing value indexes of *datapoints* are used. Structure *Point* stores location of the stimulus (which are shown in random order), intensity of that stimulus, boolean *patientsaw* for perimetry algorithm to say did patient see the point, and *committime* and *pushtime*. Commit is when program starts to show a stimulus. Push is when program stops showing the stimulus. Commit is done by calling *modelCommitPoint* and that point is pushed by *modelPushCommit*. Member *threshold* is array size of *maxthreshold* which stores intensity of the every point during after the test. Member *donebools* stores boolean value for telling is perimetry algorithm done with the point. Pointers *first*, and *last* from a linked list of patients reaction data where *first* points to first member of the linked list and *last* to last member of linked list. Member *free* points to linked list of *free ReactionDataNodes.* Nodes for linked list patient reactions are allocated in bulk and these bulks are stored there own linked list pointed by *allocmemory* which uses *ReactionDataMemory* structure which member *node* is array of

ReactionDataNodes (which is link to *free*). This linked list is mostly used when freeing the memory patient's model allocated.

Allocation of model happens in two parts to make it easier to model structure to get to different modules of the program (but there is still need for *inputbarrier* member and *modelIsSecondInitReady* function to block threads using model before second allocation is done) and give perimetry algorithm to set other information. In first allocation by *modelAllocate* model is allocated and some members initialized. Second phase is done by *modelAllocatePoints* which allocates *datapoints*, *threshold*, and *donepools* arrays to static size, and initialize amount of reaction data nodes.

Perimetry algorithm thread function is *apSignalWaiter*. This threads waits until other thread calls *apSignalToWaiter* which tells what perimetry algorithm and on what settings perimetry is run. Current implementation this thread is GUI thread function *guiStartEventLoop*. This loop handles (or otherwise calls handler) that takes in inputs from the GUI (like pressing "run test" button). Patient's input thread function is *patientInputLoop.* It is simple handle by *patientInputStart* and *patientInputEnds* which starts to gathering and end the gathering of the patient's input. Last thread handles rendering to the headset. Thread function for this is *renderingLoop.* This function uses *vrHeadsetSubmit* to submit rendered frames to the headset. Rest of the functions in *RenderingEngine* change next frame that will be rendered. Each frame renders fixation point center of the screen that patient is meant to fixate. Stimulus ordered to be rendered by *renderingStimuli* function that takes coordinates of the stimuli and intensity of it. Stimuli rendering is stoped by the *renderingResetStimuli* function. Module allows control of size of both fixation point and stimuli by *renderingScaleFixation* and *renderingScaleStimuli.* Every thread function takes in *glbExit* boolean pointer which functions as global exit signal if true. DatabaseIO module just handles writting to database (and perhaps in the future reading). It does not have it's own thread least visiable avaible. Only the GUI module and main module call DatabaseIO's function. DatabaseIO we implement was SQLite implementation for local storage of test data for bug hunting. DatabaseIO could be later be used store data to central server since interface is rather simple (just make the connection and if needed sender thread that isn't visiable

to the programmer, make your writes and then disconnect). We will not discuss too much details of this since we would have to discuss privacy.

GUI has three windows all using Windows API (Application Programming Interface). Main window where one can choose algorithm to run, press button to run, second one for grid which shows progress of the test and for debugging event window which shows events that happen during the test like committing and pushing commit point and patient's reactions (*guiAddEvent*). Event window is cleared when run is pressed. Function *guiIsFinishedCallbackForPerimetry* is callback that is called when perimetry is done. This used to have some decoupling between GUI module and rest of the system which would allow back end to be far more portable (for example same back end could be used for command line interface version of the program). Call is needed to have features like disable run button when perimetry started and enable it when perimetry is over (which doesn't happen at set time because random nature of the test). This decoupling is not perfect since function *guiClearEvent* is still called from multiple different modules. This is bit of a failing of the architecture which could be fixed by given events their own module. Image 3 show main window.

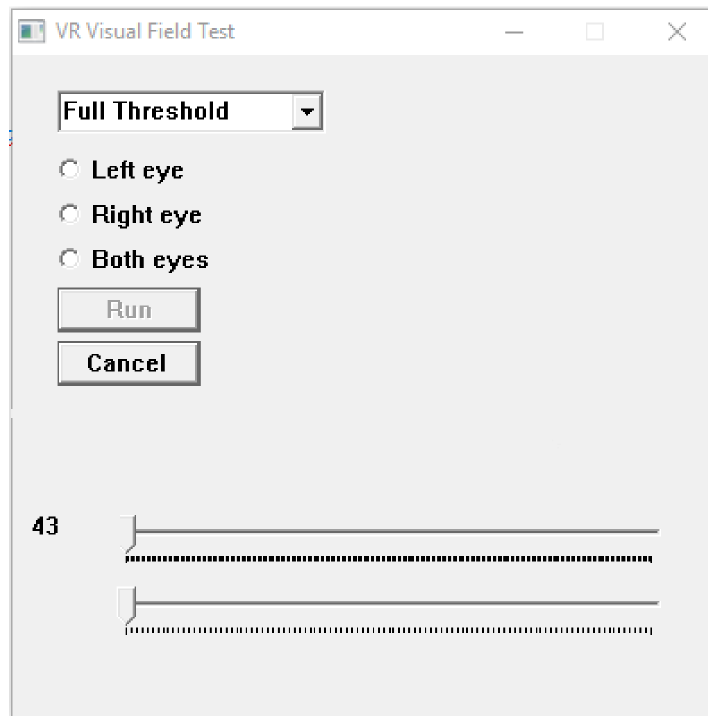

Image 3. Prototype's main window

3.2 Main in detail

Function *main* allocates memory for handles and *glbexit* and then initializes then and starts the threads:.

```
bool glbexit=false;
WindowHandler window;
VrDevice device;
guiInit(APP_NAME,&window,&glbexit);
std::thread threads[NUM_OF_ADDITIONAL_THREADS];
if(vrHeadsetDetect(&device,APP_NAME)){
      std::cout<<"Headset was not detected. Continuing anyway."
               <<std::endl;
}
Database base;
databaseCreate(&base);

threads[0]=std::thread(apSignalWaiter
                      ,&glbexit
                      ,&device
                      ,&isFinishedCallbackFromPerimetry
                      ,(void*)&window);
threads[1]=std::thread(renderingLoop,&device,&glbexit);
threads[2]=std::thread(patientInputLoop,&device,&glbexit);
guiStartEventLoop(&window,&device,&base);

apSignalCondition();
device.initcond.notify_one();
for(int i=0;i<NUM_OF_ADDITIONAL_THREADS;i++){
      if(threads[i].joinable()) threads[i].join();
      else std::cout<<"Thread not joinable!"<<std::endl;
}
vrHeadsetDeinit(&device);
renderingDeinit(&device);
closeDatabase(&base);
```

APP_NAME is a preprocessing macro that has name of the app as a string. Function *apSignalWaiter* last two parameters are callback function and argument for that callback. Since when program is start it has already one thread of execution that runs *main* we can give that thread GUI loop. GUI is chosen because it is most likely one that changes *glbexit* to true and because it doesn't have internal barriers. When program exits first it will have to make sure that no thread is in waiting mode. This is done by *apSignalCondition* and *device.initcond.notify_one*, next it will just wait for threads to join to it and then everything is freed and closed.

3.3 Automatic perimetry algorithm module in detail

Run button at the GUI will call *signalToWaiter* telling what algorithm to run and some additional information (for now you can put in eye to be tested). Insides of the *signalToWaiter* looks like following:

```
if(waiterIsWaiting){

      funcIndex=funcindex;
      currentEye=eye

      // Allocate new model
      if(currentmodel) delete currentmodel;
      currentmodel=new Model();

      // Signal about waiting condition change
      isWaiting=false;
      return currentmodel;
}
return 0;
```

By returning new model to caller *VrVisualFieldGui* will get copy of the model and hence ability to query data from the model. Function *apSignalWaiter* implementation is:

```
std::unique_lock<std::mutex> lock(condVarMutex);
if(device->initdone){
      while(glbexit){
            waiterIsWaiting=true;
            condVar.wait(lock,[]{return !waiterIsWaiting;});
            if(*glbexit) break;
            clearEvent();

            clock_gettime(CLOCK_MONOTONIC,
                          &currentmodel->startingtime);
            guiAddEvent("Starting perimetry!"
                        ,&currentmodel->startingtime
                        ,"");
            funcs[funcIndex](device,glbexit);
            patientInputEnds();

            isfinished(isfinisheddata);
      }

}
```

Note that we don't start taking input from the patient just yet. That responsibility is moved to inside of the perimetry algorithm we decided to run. This is because two step initialization of the model and hence when we call perimetry algorithm we haven't finished initialization.

In Full Threshold the second initialization phase looks like following:

```
std::srand(std::time(NULL));
const int tableofpointssize=sizeof(TableOfPoints)
                            /sizeof(TableOfPoints[0]);
modelAllocatePoints(26*tableofpointssize
                   ,20*tableofpointssize
                   ,tableofpointssize
                   ,currentmodel);
patientInputStart(currentmodel);
for(uint32_t i=0;i<tableofpointsize;i++){
      currentmodel->threshold[i]=18;
}
```

First line initialize random generation seed. Second line is just constant that is used a lot. It tells us how many points *TableOfPoints* has. Global variable *TableOfPoints* is array of coordinates 30-2 grid is located in the our screen. Third, fourth and fifth line uses constant to allocate points 22 times the constant, 26 times the constant patient reactions for start and constant's number of thresholds. First coefficient comes from that every test point starts at intensity of 18 [2] (last line) which increase or decrease in jumps of four and at max taking six steps to reach 40 then potentially changing direction to other end in jumps of two making 20 steps. Second coefficient comes from same type of reasoning but unlike previously 20 would be enough since first six would be patient not reacting to stimuli. Last parameter needs only be constant amount since that constant is *tableofpointssize* meaning one per point in 30-2 grid.

When initialization is done Full Threshold implementation goes to do while loop. Loop ends if ether global exit is called, perimetry is cancelled or if there is no points to be shown. Following code checks that there isn't any points to be show.

```
run=false;
uint32_t c=0;
for(uint32_t i=0;i<sizeof(pointsshow);i++){
      if(pointsshow[i]){
            run=true;
            c++;
      }
}
userecent=(c>sizeof(pointsshow)-sizeof(recent)/sizeof(recent[0])-1);
```

Variable *run* has to survive as false the for loop if there is no points to show. Array *pointsshow* has information of how many directional changes each point of the grid has left. If *pointsshow*

indexes are zero then there is no more points to show making if sentence in the for loop never executing. Reason if sentence doesn't have break is to calculate *c* variable. It is used to calculate *userecent* which is true if *c* (number of points that could be shown next run) is bigger then number of points (first *sizeof*) minus the number of points *recent* array stores. Array *recent* is used in commit point (next stimulus) selection to prevent point being shown right after it was shown unless there is too few points to be picky with the selection. Following code handles committing and *recent* array.

```
do{
      DO_WHILE_JUMPBACK:
      index=std::rand()%tableofpointssize;
}while(pointsshow[index]==0);
if(userecent){
      for(uint8_t check=0;
          check<sizeof(recent)/sizeof(recent[0]);
          check++){
            if(recent[check]==index) goto DO_WHILE_JUMPBACK;
      }
}
runMilliSecondSpin(200+std::rand()%800,last);
clock_gettime(CLOCK_MONOTONIC,&last);
modelCommitPoint(device
                ,TableOfPoints[index].x
                ,TableOfPoints[index].y
                ,intensity[index]
                ,currentmodel);
for(uint8_t mv=0;mv<sizeof(recent)/sizeof(uint32_t)-1;mv++){
      recent[mv]=recent[mv+1];
}
recent[sizeof(recent)-1]=index;
```

Do while randomly selects index of the next point from the *TableOfPoints* array long as *pointsshow* isn't zero. When execution comes out of do while if boolean *userecent* is true for loop is used to check that *index* isn't in *recent* array. If *index* is found in the recent then execution is jumped back to do while loop. After we are certain that *index* is usable then make sure 200 to 1000 milliseconds has been taken from last time was taken. Variable *last* stores comparison clock value and it is taken before commit, before input processing and after push of a point. Time interval is randomly selected to trick human mind not to expect the stimuli on certain time interval. We don't just wait certain time interval because processing can be done into this waiting time. After the wait point is committed *recent* array is updated in the last lines. We can use *std::rand* for pseudo random generator as we don't use it for security but tricking

human mind that there is no pattern. Support function *runMilliSecondSpin* uses *TimeOp* module to that asks when second parameter plus first parameter is more or equal to current time. We experimented with sleeping however this was more inaccurate then just spinning.

After this we check that 200 milliseconds have passed before pushing stimuli and retake *last*. Also note that *recent* arrays update goes into this 200 milliseconds. Timer here is at human reaction time limit and could be smaller or bigger depend on patient.

```
runMilliSecondSpin(200,&last);
modelPushCommit(device,currentmodel);
clock_gettime(CLOCK_MONOTONIC,&last);
```

Next patient's reaction to stimuli is handled by first giving 600 milliseconds to patient to react. This time goes to amount stimuli isn't show making time between stimuli 800 to 1600 milliseconds.

```
runMilliSecondSpin(600,&last);
Point *point=current->datapoints+current->size-1;
while(processingpoint->next
  && timeOpCmp(&point->committime,&processingpoint->next->timestamp)){
      processingpoint=processingpoint->next;
}
struct timespec memoryforaddition;
while(processingpoint->next
  && timeOpCmp(&processingpoint->next->timestamp,&point->committime)
  && timeOpCmp(timeOpAddConstMs(&point->pushtime
                                ,600
                                ,&memoryforaddition))){

      processingpoint=processingpoint->next;
      if(processing->classification==PT_IN_SPACE){
            point->patientsaw=true;
      }
}
```

First while processes *processingpoint* variable to where last show stimuli (*point* variable) so that we checking only reactions that could have happened during stimuli. Variable *processingpoint* is pointer to *ReactionDataNode* and it is set at start to point to beginning of the model's member *first*. During reaction processing *processingpoint* moves through linked list of reaction. Next while loop checks does every reaction between time point was committed and 600 milliseconds from commit. If reaction happens between and it is pressing of the space bar it is consider that patient saw the point. Function *timeOpCmd* returns true if first *timespec* is younger or equal to second *timespec*.

After algorithm has checked patient reactions variable *last* is updated and we start to handle what patient reaction or lack of reaction means. We check does *pointsshow[index]* need to be updated.

```
if(pointsshow[index]==3){
      firstseen[index]=point->patientsaw;
      pointsshow[index]--;
}
else{
      bool nextif1=(pointsshow[index]==2
                  && (point->patientsaw!=firstseen[index]));
      bool nextif2=(pointsshow[index]==1
                  && (point->patientsaw==firstseen[index]));
      if(nextif1 || nextif2) pointsshow[index]--;
}
```

As said earlier array *pointsshow* has information of how many directional changes each point of the grid has left. At beginning *pointsshow*'s indexes are initialized to three and it means no point is shown at the point yet. When stimuli is shown on unshown point then whatever patient saw the stimuli is recorded to *firstseen* array and *pointsshow[index]* is set to two. Array *firstseen* is used to know what was starting direction of the stepping on each point. Next time *pointshown* on index is reduced if at state 2 *firstseen* isn't same at what *patientsaw* (patient saw first stimuli but didn't see it now or patient didn't saw but now did) and at state 1 if they are equal (patient saw first stimuli and saw one now and patient didn't see and didn't see now).

After this we have decide what direction intensity should go next on this point if *patientsaw* didn't get to zero which point we are done with the point.

```
if(0<currentmodel->threshold[index]
&& currentmodel->threshold[index]<40){
      switch(pointsshow[index]){
            case 2:
                  if(firstseen[index]){
                        if(currentmodel->threshold[index]-4<0){
                              currentmodel->threshold[index]=0;
                        }
                        else currentmodel->threshold[index]-=4;
                  }
                  else{
                        if(currentmodel->threshold[index]+4>40){
                              currentmodel->threshold[index]=40;
                        }
                        else currentmodel->threshold[index]+=4;
```

```
                        }
                        break;
                  case 1:
                        if(firstseen[index]){
                              if(currentmodel->threshold[index]+2>40){
                                    currentmodel->threshold[index]=40;
                              }
                              else currentmodel->threshold[index]+=2;
                        }
                        else{
                              if(currentmodel->threshold[index]-2<0){
                                    currentmodel->threshold[index]=0;
                              }
                              else currentmodel->threshold[index]-=2;
                        }
                        break;
            }
      }
      if(0==currentmodel->threshold[index]
        || currentmodel->threshold[index]>=40){
            pointsshow[index]=0;
      }
      if(pointsshow[index]==0) current->donebools[index]=true;
```

Since we reduce *pointsshow[index]* before we get here case for three is not needed since it would have been reduced. At case two we move 4 step (decibels) and in 2 steps in case one. Otherwise both states do similar calculations. If first stimuli was seen we reduce intensity for next step on state two and increase it on state one and reverse the directions if first stimuli wasn't seen. We check that intensity is kept in zero to 40 range. Zero being most intensive we going to output. After we have changed intensity we check did we hit zero or 40 and if we did we just mark that point is done since there is no next staircase to move.

3.4 Model operations in detail

Function *modelAllocate* is simplify allocates the model and initialize some of the members of the structure:

```
Model *model=new Model;
model->size=0;
model->maxsize=0;
return model;
```

Model's second allocation phase has far more information that is handled. Parameters given in to *modelAllocatePoints* are *pointsize* (maximum number of stimuli shown), *reactionsize*

(initialize number of patient reactions), *threshold* (number of grid points), *model* (model we are second initialize).

```
model->datapoints=new Point[pointsize];
model->maxsize=pointsize;
model->allocmemory=new ReactionDataMemory;
model->allocmemory->next=NULL;
model->allocmemory->node=new ReactionDataNode[reactionsize];
model->free=model->allocmemory->node;
{
      int32_t i=0;
      while(i<reactionsize-1){
            model->free[i].next=model->free+i+1;
            i++;
      }
      model->free[i].next=NULL;
}
model->last=(ReactionDataNode *)&model->first;
model->first=0;
model->threshold=new uint32_t[thresholds];
model->maxthreshold=threshold;
memset(model->threshold,1,thresholds);
model->donebools=new uint8_t[thresholds];
memset(model->donebools,false,thresholds);
```

We reset *maxsize* to actual number of points here so between allocation function *maxsize* is used as state indicator in *modelIsSecondInitReady*. We allocate first stack of *ReactionDataNode*s which we then initialize (while loop sets the linked list of free reactions nodes). Small optimization done here is set *last* to equal address of the *first*. Since *ReactionDataNodes* first member is next pointer we can "trick" *modelAddReaction* to think first node already exist in the link so we can omit if clause for checking are we adding first node to list. Model is destroyed by *modelDelete* which just takes in the mode to destroy and just deletes all of the arrays allocated and goes through *ReactionDataMemory* linked list deleting *ReactionDataNode*s array and then deleting *ReactionDataMemory* itself. After this is done *Model* structure itself is deleted.

Stimuli is committed by *modelCommitPoint* which takes in intensity and coordinates (*updown* and *temporalnasal*).

```
if(model->size<model->maxsize){
      CommitMutex.lock();
      Commit=&model->datapoints[model->size];
      Commit->intesity=intensity;
      Commit->updown=updown;
```

```
        Commit->temporalnasal=temporalnasal;
        Commit->patientsaw=false;
        clock_gettime(CLOCK_MONOTONIC,&commit->committime);
        CommitMutex.unlock();
        renderingStimuli(device,temporalnasal,updown,1-intensity/40.0f);
        char extrstr[12];
        sprintf(extrstr,"(%d,%d)",updown,intensty);
        addEvent("Commit",&Commit->committime,extrstr);
}
```

We check that we don't overflow after which get the lock so that other threads don't read incomplete data and fill data to *datapoints* index where *size* is at. Global pointer *Commit* is used to tell other threads there is stimuli shown if it is non null and what data it has (functions *modelGetCommitLock* for getting the commit and *modelReleaseCommitLock* release the lock). We also make call to *renderingStimuli* where to render the stimuli. We have to change intensity from 0 to 40 values where 0 was highest intensity to 0.0 to 1.0 decimal number where 1.0 is highest intensity. Division by 40 does the normalization to 0.0 to 1.0 range and 1 minus does the flip of high intensity other end of the range.

Incrementing of the *size* member of the *Model* happens in at *modelPushCommit* which orders stimuli to stop render and changed *Commit* to null.

```
resetStimuliPoint(device);
CommitMutex.lock();
clock_gettime(CLOCK_MONOTONIC,&Commit->pushtime);
Commit=NULL;
CommitMutex.unlock();
model->size++;
```

3.5 Rendering in detail

Rendering in current implementation is done by Vulkan. Shaders (piece of GPU program in the graphics pipeline) are very simple. Fragment shader has nothing special in it but vertex shader has some code for scaling stimuli or fixation point and calculating correct location. Variable *gridscaler* is scaling variable translating between visual field angels and pixel coordinates at the screen. Current value of *gridscaler* is rough estimation.

```
#version 450
#extension GL_ARB_separate_shader_objects : enable
layout(binding=0) uniform _display{
      vec2 pointscaler;
}display;
float gridscaler=0.09;
```

```
layout(location=0) in vec2 pixels;
layout(location=1) in vec2 position;
layout(location=2) in float intensity;
out gl_PerVertex {
    vec4 gl_Position;
};
layout(location = 0) out vec3 fragColor;

void main(){
      float x=display.pointscaler.x;
      float y=display.pointscaler.y;
      float z=0;
      mat4 scale=mat4(vec4(x,0,0,0)
                     ,vec4(0,y,0,0)
                     ,vec4(0,0,z,0)
                     ,vec4(0,0,0,1));
      gl_Position=scale*vec4(pixels,0.0,1.0)
                 +vec4(gridscaler*position,0.0,1.0);
      fragColor=vec3(intensity,intensity,intensity);
}
```

Both stimuli and fixation use same shader pipeline (GPU (Graphics Processing Unit) program) to render. Rendering is done to two frames one for each eye. These are represented by *VkImage* Vulkan objects in *VrDevice* by *frameimage* array. We create two *VkCommandBuffer* object (which are list of high level instructions to GPU) once in *renderingInit*. They are stored also in *VrDevice* structure as *cmdbufffixation* and *cmdbuffstimulus*. Both command buffers don't have subpasses. Command buffer for fixation point clears the both frames in *frameimage* array. This means that fixation point command buffer should always be executed before stimulus command buffer. Fixation and stimulus command buffers both bind similar resources (after all they use same shaders) for upcoming draw calls. Command buffer for fixation binds following:

```
vkCmdBindDescriptorSets(device->cmdbufffixation
                        ,VK_PIPELINE_BIND_POINT_GRAPHICS
                        ,device->pipelinelayout
                        ,0
                        ,1
                        ,&device->descsetfixation
                        ,0
                        ,0);
const VkDeviceSize offsets[1]={0};
vkCmdBindVertexBuffers(device->cmdbufffixation
                       ,0
                       ,1
                       ,&device->vertexbuffer
                       ,offsets);
vkCmdBindVertexBuffers(device->cmdbufffixation
                       ,1
                       ,1
```

```
                            ,&device->instancebuffer
                            ,offsets);
```

Description set is where shader get *pointscaler* vector used in scaling which is given value fixation buffers description set (*descsetfixation*). Description set is not meant to be changed while test is going on. First *vkCmdBindVertexBuffers* binds vertex data to vertex shader's *pixel* which is the two dimensional model of fixation target. Right now this buffer is rough circle that is used in both command buffers. Last *vkCmdBindVertexBuffers* call binds instance buffer that used to tell position and intensity of point to shader's *position* and *intensity* variables. Difference between the two binds is that shaders are executed for each pair in *vertexbuffer* and *instancebuffer* contest is kept same for each (more detail look at Vulkan speficication). For *cmdbuffstimulus* binds differ only in that stimulus has it's own scaling description set and that instancebuffer is used from different offset. Variable *pipelinelayout* from *VrDevice* is layout information of graphics pipeline (essentially it has information regarding locations and binding numbers in the shader like memory size and vertex or instance buffer).

Scaling buffers are changes by calling *renderingScaleStimuli* and *renderingScaleFixation* which both inners are the same but description set is different. They both take in array of two floats called *vec2* and the device scaling will change.

```
    if(device->initdone){
          device->renderaccess.lock();
          float *data;
          vkMapMemory(device->logicaldevice
                      ,device->descsetgpumem
                      ,0
                      ,sizeof(float)*2
                      ,0
                      ,(void **)&data);
          data[0]=vec2[0]/1200;
          data[1]=vec2[0]/1200;
          vkUnmapMemory(device->logicaldevice,device->descsetgpimem);
          device->renderaccess.unlock();
    }
  }
```

Rendering access is halted while operation is made to make sure that GPU side memory isn't changed during execution. Variable *logicaldevice* is *VkLogicalDevice* handle (reference to GPU in virtual manner). During *renderingInit* memory for description set is initialized

(*descsetgpumem*) we mapping temporally to host memory (CPU side) and then edit the data and undo the mapping changing data at GPU side. We divide *vec2* with 1200 so that GUI could have simple slider with integer numbers when in reality virtual reality headset coordinated are in -2.0 to 2.0 range.
Actual commands to draw differ in that fixation point is rendered to both frames where stimulus is rendered only to first buffer. Benefit rendering fixation point on both frames and stimulus to only one is that you don't eye patch which traditional machines need. Patient's eyes get there own images which patient will not be aware of. Draws are made by simple *vkCmdDraw* with Vertex buffer pair amount time one instance starting at zero offset on both vertex and instance buffers.

Function *renderingStimuli* insides change GPU side memory similarly to scaling functions but it also set *numberofactivebuffers* member of *VrDevice* to two.

```
InstanceData *insdata;
vkMapMemory(device->logicaldevice
           ,device->vrinstancememory
           ,sizeof(InstanceData)
           ,sizeof(InstanceData)
           ,0
           ,(void **)&insdata);
insdata->position[0]=x;
insdata->position[1]=y;
insdata->intensity=intensity;
vkUnmapMemory(device->logicaldevice,device->vrinstancememory);

device->numberofactivebuffers=2;
```

When push of the stimulus comes *renderingResetStimulus* is called and it just sets *numberofactivebuffes* to it's initialized value of one. On *renderingLoop* we have local array *activebuffers.*

```
VkCommandBuffer activebuffers[2]={device->cmdbufffixation
                                 ,device->cmdbufflefteye};
```

When we call *VkQueueSubmit* (function that sends command buffers to) in our loop after locking *renderaccess VkSubmitInfo* structure look following.

```
VkPipelineStageFlags waitstage[1];
waitstage[0]={VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT};
VkSubmitInfo submitinfo;
submitinfo.sType=VK_STRUCTURE_TYPE_SUBMIT_INFO;
submitinfo.pNext=NULL;
```

```
submitinfo.commandBufferCount=device->numberofactivebuffers;
submitinfo.pWaitDstStageMask=waitstage;
submitinfo.pWaitSemaphores=NULL;
submitinfo.pWaitSemaphoresCount=0;
submitinfo.pSignalSemaphores=NULL;
submitinfo.signalSemaphores=0;
result=VkQeueueSubmit(device->queue,1,&submitinfo,VK_NULL_HANDLE);
if(result!=VK_SUCCESS) std::cerr<<"Submit error: "<<result<<std::endl;
```

Interesting thing here is we can send multiple command buffers at the same time to the execution queue (*queue*). We can control this number by directly *numberofactivebuffers* so if *numberofactivebuffers* is one only *cmdbufferfixation* is putten to execution when *numberactivebuffes* is two we send both command buffers and there order is kept the same.

After command buffers have being executed we have two *VkImage*s which both have fixation point and first one may have stimulus rendered to it. We then call *vrHeadsetSubmit* which takes in the *VrDevice*.

vrHeadsetSubmit looks like following.

```
vr::VRVulkanTextureData_t vulkandata;
vulkandata.m_nImage=(uint64_t)device->frameimage[0];
vulkandata.m_pDevice=device->logicaldevice;
vulkandata.m_pPhysicalDevice=device->physicaldevice;
vulkandata.m_pInstance=device->instance;
vulkandata.m_pQueue=device->queue;
vulkandata.m_nQueueFamilyIndex=device->queuefamily;
vulkandata.m_nWidth=device->renderwidth;
vulkandata.m_nHeight=device->device->renderheight;
vulkandata.m_nFormat=VK_FORMAT_R8R8G8A8_UNORM;
vulkandata.m_nSampleCount=1;
vr::Texture_t submitimage={&vulkandata
                          ,vr::Texture_Vulkan
                          ,vr::ColorSpace_Auto};
vr::VRTextureBounds_t bounds;
bounds.uMin=0.0f;
bounds.uMax=1.0f;
bounds.vMin=0.0f;
bounds.vMax=1.0f;
```

Implementation is for OpenVR API so we have to fill data structure that tells OpenVR that frames are Vulkan objects. Then we query what eye we want stimulus appear have following switch case:

```
vr::EVRCompositorError errorleft,errorright;
switch(apGetEye()){
      case VREYE_LEFT:
            errorleft=vr::VRCompositor()->Submit(vr::Eye_Left
```

```
                                                            ,&submitimage
                                                            ,&bounds);
                    vulkandata.m_image=(uint64_t)device->frameimage[1];
                    errorright=vr::VRCompositor()->Submit(vr::Eye_Right
                                                            ,&submitimage
                                                            ,&bounds);

                    break;
              case VREYE_RIGHT:
                    errorright=vr::VRCompositor()->Submit(vr::Eye_Right
                                                            ,&submitimage
                                                            ,&bounds);
                    vulkandata.m_image=(uint64_t)device->frameimage[1];
                    errorleft=vr::VRCompositor()->Submit(vr::Eye_Left
                                                            ,&submitimage
                                                            ,&bounds);
                    break;
        }
```

Depend on what eye is returned by *apGetEye* case will select what order image are submitted to OpenVR's compositor. This way we don't have third command buffer which renders same then stimulus command buffer but on second index of *frameimage*.

4 IN CONCLUSION

We have discussed how to make basic implementation of program that will test patient's visual field on Virtual Reality headset. We have given basic architecture for this implementation which isn't only or best architecture out there (back end modules are depended on GUI module). Important feature missing for program is program reacting patient's eye moving away from center when stimulus is on screen or patient blinks but these detection can be done with Fove's eye tracking capability. There is also more research needed to know how to translate up down and temporal nasal angel coordinate to pixels on screen and knowing exactly how much output VR headset gives.

REFERENCES

[1] Cubbidge, R. 2005. Visual Fields.Elsevier Health Sciences.
[2] Racette, L., Fischer, M., Bebie, H., Holló, G., Johnson, C. A., & Matsumoto, C. . Racette, Lyne, et al. "Visual field digest." A guide to perimetry and the Ocotpus perimeter.Haag-Streit AG.
[3] Wroblewski, D., Francis, BA., Sadun, A., Vakili, G., & Chopra, V.. 2014.Testing of Visual Field with Virtual Reality Goggles in Manual and Visual Grasp Modes.Biomed Res International, 206082. http://doi.org/10.1155/2014/206082.
[4] Tsapakis, S. Papaconstantinou, D. Diagourtas, A. Droutsas, K. Andreanos, K. Moschos, MM. & Brouzas D. 2017. Visual field examination method using virtual reality glasses compared with the Humphrey perimeter. Clinical Ophthalmology. Vol.11, p.1431-1443
[5] Boel B., Jonny, O., Anders, H., & Holger, R. 1997. A new generation of algorithms for computerized threshold perimetry, SITA. Acta Opthalmologica Scandinavica. Vol.75, issue 4, p.368-375